Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Languages and Types for Component-Based Programming

João Ricardo Viegas da Costa Seco

Dissertação apresentada para a obtenção do Grau de Doutor em Informática pela Universidade Nova de Lisboa, Faculdade de Ciências e Tecnologia.

Lisboa
(2006)

Evaluation Committee:

Luís Monteiro, Universidade Nova de Lisboa (President)
Sophia Drossoupoulou, Imperial College London (Main Referee)
Davide Ancona, Universita di Genova (Main Referee)
Luís Caires, Universidade Nova de Lisboa (Supervisor)
Antónia Lopes, Universidade de Lisboa
Luís Lopes, Universidade do Porto
Artur Miguel Dias, Universidade Nova de Lisboa

This dissertation was prepared under the supervision of
Professor Luis Caires,
of the Faculdade de Ciências e Tecnologia,
Universidade Nova de Lisboa.

To my loving wife Joana, and my lovely children Martim, Teresa, and Leonor

Acknowledgements

The work presented in this dissertation would not have been possible without the collaboration of a considerable number of people to whom I would like to express my gratitude.

To my wife Joana, for her dedication and unconditional love. To my son Martim, and my two daughters, Teresa and Leonor, for their everlasting love and tenderness. I stole too much of the time that was rightfully theirs in these past years. To my parents, Carlos and Zulmira, for their confidence, and also to my parents in law, Ricardo and Lalita, for their continuous encouragement and support.

To Luís Caires to whom I owe a great deal, for his guidance on a area of research which was completely new to me, for the long hours discussing from the most abstract ideas to the finest of details. His example and encouragement were of most importance to achieve the present result.

To Sérgio Duarte and Henrique João Domingos with whom I shared an office for many years and who showed infinite patience for all the gibberish that constantly filled our whiteboard. To Anabela Ribeiro, Carmen Morgado, Cecília Gomes, Fernando Birra, João Lourenço, Nuno Preguiça, Pedro Medeiros, and Vítor Duarte for being the best of colleagues. To Miguel Durão for experimenting with componentJ and giving a hand with the ML implementation of the component calculus.

To Luís Monteiro, coordinator of the ”Programming Languages and Models” Research Stream of the CITI, for his encouragement and support. To all the anonymous referees that reviewed our papers and commented on our research approaches and results. To Luca Cardelli for some discussions on components and dynamic reconfiguration, and to Dario Colazzo for his comments on our work about subtyping recursive types.

My work was partially funded by the Fundação para a Ciência e Tecnologia, by the research stream Programming Languages and Models of the Centro de Investigação em Informática e Tecnologias da Informação research centre (CITI), by the research project Databricks (POSI/33924/00), and the ComponentGlue project funded by Microsoft Research Cambridge (MSR Research Grant 2002/73).

Abstract

Component-Oriented Programming has been, for the past few years, an emergent programming paradigm. However, it has been supported mainly by low-level technological solutions. Traditionally, modularity in object-oriented languages has been based on the notion of class, and software reuse achieved by either implementation inheritance or object aggregation. On the one hand, inheritance has proved to become problematic in the context of large-scale software systems, as it may hinder evolution and interferes with issues such as dynamic loading. On the other hand, object aggregation is usually implemented by ad-hoc idioms that manually build webs of objects.

This dissertation contributes to this field by introducing appropriate programming language abstractions and type systems for expressing the assembly, adaptation, and evolution of software components. Our model is formally expressed by means of a core typed programming language whose first-class values are objects, components, and configurators. *Objects* are component instances which aggregate state and functionality in the standard object-oriented sense. *Components* are entities that specify the structure and behaviour of objects by means of a combination and adaptation of smaller components. From a network of elements, which is specified by a configurator value, only well identified interfaces, that import and export services, remain visible to be used in further compositions. *Configurators* are operations that produce structures by aggregating and connecting components in an implementation independent way; they are either defined using canonical composition operations, that insert or connect elements in a composition, or by a combination of other configurators, thus producing their joint effect. Configurators are uniformly used to produce components or modify the internal structure of objects. Thus, this variety of values and language constructs allows for both the expression of dynamic construction of new components (based on run-time decisions) and the unanticipated reconfiguration of component instances.

We develop several type systems to ensure the structural soundness of components and objects, before and after reconfiguration actions. In particular, configurators are typed with intensional type information, revealing certain aspects of their internal structure which play an essential role in the typing of composition and reconfiguration actions. We also define a novel approach to subtyping second-order recursive types, we prove its decidability by designing a subtyping algorithm and proving its correction. This result is then used to enrich our component language with recursive types and polymorphism (universal and subsumption-based), and define a flexible structural subsumption relation between components, instances and reconfiguration scripts. The decidability of the typing relation and the correction of the corresponding algorithms are also presented and proved correct. We define and prove type safety by means of subject reduction properties which imply the intended structural soundness results.

To perform actual experimentation of our programming model, we also developed and implemented a prototype compiler for a Java-like programming language, componentJ.

Sumário

A Programação Orientada por Componentes tem tido nos últimos anos uma relevância crescente como paradigma de programação. No entanto, tem sido apenas suportada por soluções tecnológicas de baixo nível. Tradicionalmente, a modularidade em linguagens orientadas por objectos tem sido baseada na noção de classe e a reutilização de software conseguida através de mecanismos de herança ou de agregação de objectos. Por um lado, o mecanismo de herança pode tornar-se problemático no contexto do desenvolvimento de software em larga escala pois condiciona a evolução dos programas e interfere com outros mecanismos como por exemplo o do carregamento dinâmico de código. Por outro lado, a agregação de objectos é normalmente implementada por redes de objectos montadas de uma forma ad-hoc ou seguindo alguns idiomas de programação.

A contribuição desta dissertação está na introdução de abstracções adequadas à definição da montagem, adaptação e evolução de componentes de software ao nível das linguagens de programação. O nosso modelo é expresso formalmente numa linguagem de programação cujos valores de primeira classe são objectos, componentes e configuradores. Os *objectos* são instâncias de componentes que agregam estado e funcionalidade como é habitual nas linguagens orientadas por objectos. Os *componentes* são entidades que especificam a estrutura e comportamento dos objectos. A sua construção é feita por uma operação da linguagem que, a partir de um configurador que liga e adapta outros componentes, produz um valor que esconde os detalhes da sua estrutura interna. Apenas os pontos de ligação explicitamente definidos, denominados portos, ficam visíveis no componente resultante. Os *configuradores* são operações que produzem estruturas por agregação e conexão de componentes independentemente das suas implementações. Os configuradores são definidos a partir de operações canónicas de composição que inserem ou ligam elementos, ou por composição de outros configuradores. Os configuradores são aplicados de uma maneira indiferenciada tanto na produção de componentes como na modificação da estrutura interna de objectos. Esta panóplia de valores e abstracções permite a codificação tanto da construção dinâmica de novos componentes (tendo como base decisões de execução), como a reconfiguração inesperada de instâncias de componentes.

São apresentados nesta dissertação vários sistemas de tipos que asseguram a consistência estrutural de componentes e objectos, antes e depois de acções de reconfiguração. Em particular é utilizada informação de tipos intensional para descrever os configuradores que revela aspectos da sua estrutura interna. Esta informação desempenha um papel fundamental na tipificação de operações de composição e de reconfiguração. Também se define uma relação inovadora de subtipificação para tipos recursivos de segunda ordem e é apresentada e provada a correcção do algoritmo correspondente. A linguagem de componentes é então extendida com tipos recursivos e polimorfismo (paramétrico e de inclusão) utilizando estes resultados. A decidibilidade da relação de tipificação e a correcção do algoritmo correspondente são também demonstradas. A segurança de tipos é apresentada e expressa formalmente num resultado de *subject reduction* que implica os resultados de consistência estrutural desejados.

Para experimentar o modelo, foi desenvolvido e implementado um protótipo de um compilador para uma linguagem de programação baseada em componentes chamada componentJ, que tem uma sintaxe semelhante à linguagem de programação Java.

Notation

Expressions	e, e', e_i
Labels	ℓ, ℓ', ℓ_i
Set of labels	$\mathcal{L}$
Variables	$x, y, z, c, d, f, g, \ldots$
Set of variables	$\mathcal{V}$
Locations	$l,\ l_i,\ l'$
Set of locations	$\texttt{Loc}$
Values	$s, t, u, v, \ldots$
Set of values	$\mathcal{U}$
Memory heaps	$S,\ S_i,\ S'$
Ports	$o, p, q, r, \ldots$
Records	$r,\ r',\ r_i$
Empty instance	$\mathbf{0}$
Types	$\tau,\ \sigma,\ \delta,\ \gamma,\ \alpha,\ \beta,\ \tau',\ \tau_i$
Set of types	$\mathcal{T}$
Type variables	X, Y, Z, W
Set of type variables	$\mathcal{Z}$
Substitution of variables	θ
Unitary substitution	$[x \leftarrow e']$
Substitution of type variables	Θ
Unitary substitution of type variables	$[X \leftarrow \tau]$
Restriction of a name x in a substitution	$\theta \backslash \{x\}, \Theta \backslash \{X\}$
Sequences	$e_i\ ^{i \in 1..n} \triangleq e_1, \ldots, e_n$
Interface types	$I,\ J$
Object types	$R,\ P$
Component Types	$R \Rightarrow P$
Resources	$\ell \bullet \tau, \ell \circ \tau, \ell \triangleright \tau, \ell \triangleleft \tau$
Resource sets	K,K',K_i

Configurator types	$K \Longrightarrow K'$
Typing environments	$\Delta, \Gamma, \Pi, \Delta', \Delta_i$
Restricted typing environment	$\lvert\Delta\rvert$
Well-formed type judgement	$\Delta \vdash \tau\ \mathsf{ok}$
Valid typing environment judgement	$\Delta \vdash \diamond$
Typing judgement	$\Delta \vdash e : \tau$
Subtyping judgement	$\Delta \vdash \tau \leq \sigma$
Evaluation judgement	$e; S \downarrow v; S$
Configurator application judgement	$s; e; S \Downarrow s; S$
Matching of an instance to a resource set	$s \mathbin{/\!/} K$

Concatenation of records	$r \oplus r'$
Concatenation of interfaces	$I \oplus J$
Concatenation of resource sets	K, K'
Disjointness of resource sets	$K \# K'$
Disjointness of interfaces	$I \# J$
Unsatisfied resources in K or s	$K_\circ, s_\circ$
Available resources in K or s	$K_\bullet, s_\bullet$
Provided ports in K or s	$K_\triangleright, s_\triangleright$
Required ports in K or s	$K_\triangleleft, s_\triangleleft$

Narrowing relation on typing environments	$\Gamma \sqsubseteq \Gamma$
Generating function for a subtyping relation	S
Greatest fixed point of a function S	$gfp(S), \nu S$
Tuples of a subtyping relation	$(\Delta, \tau, \sigma), t, t'$
Similarity relation on tuples	$t \simeq t'$
Reachability relation on tuples	$t \gg t'$
Subexpression relation on types	$\tau \preceq \tau'$

Contents

List of Figures

Chapter 1

Introduction

Component-Oriented Programming is an emerging programming paradigm for which appropriate programming language support still seems to be lacking. Its main goal, namely to combine prefabricated software components in ways such that reuse is made possible at a global scale, has been supported mainly by low-level linking mechanisms such as JavaBeans, COM, and .NET.

1.1 Modularity and Programming Languages

Traditionally, modules have been the usual mechanism provided at the programming language level for reusing code and taming the structural complexity of software systems. From the rudimentary support for linking of code fragments in the C programming language [62] to the sophisticated module language of ML [70], there is a large variety of approaches allowing programmers to build applications from separately developed and compiled blocks of code. Modules enforce information hiding at the level of system components. Consistency between different parts of a software system is then ensured using interfacing information revealing public module elements. The process of actually building executable systems is usually expressed by compilation and linking procedures such as *makefiles*. Nevertheless, some programming environments do allow the assembly of systems to be specified in a high-level composition language. For example, in ML, modules may be composed using a notion of functor, that essentially corresponds to a parameterised module. In all cases, the structure of systems must be fully resolved at compile-link-time and gets fixed once and for all.

Module languages provide compile-time soundness guaranties over the structures they describe. However, they fail to capture global dependencies between modules, those that end up implicit in their implementations. An exception can be found in the ML functor language where functors explicitly indicate inter-module dependencies. Additionally, the majority of module systems only support strictly hierarchical dependencies. The theory of modules has been widely studied by authors such as MacQueen, Harper and others [70, 54, 16, 53, 67, 44, 33, 10]. In particular, Cardelli [24] and Leroy [68] developed some work that describes and formalises properties of general linking processes. More recently, some type safe approaches to module composition, usually called Mixin-modules, were studied by several authors [16, 38, 101, 11, 56, 57]. Mixin-module languages separate the definition of modules, using a base language, from their composition, which is achieved by means of operators on modules. These operators manipulate the imported and exported names of its operands and in this way combine their implementations to obtain larger modules. Mixin-modules provide a more flexible composition mechanism for combining modules than the traditional hierarchical approach, and to some extent they inspired some of our work. In particular, they allow the definition of structures with mutually dependent modules.

The structuring mechanisms introduced by object-oriented programming, allowing the programmer to group related data and functionality using classes, is the commonly accepted way of modelling application scenarios and, at a small scale, modularise software systems. However, classes are not just units of modularity, they are also units of inheritance-based code reuse. It is currently believed that the strongly-coupled structures induced by implementation inheritance do not mix well with the needs of software evolution, even less if combined with dynamic linking mechanisms such as the ones introduced by modern run-time environments such as Java and .NET. Inheritance hinders the evolution of class implementations; it is known that any modification to a class that is not a leaf in the inheritance hierarchy may cause code to break in other parts of the system. Thus, large scale object-oriented applications are prone to the so-called *fragile base class problem* [46, 97].

Alternatively, when one chooses to compose elements by aggregation instead of class extension, the (client/server) relations between objects are usually represented by references stored in instance variables. Unless a careful programming discipline is followed, these manually built object structures, which are commonly hardwired in the source code, are error prone, hard to maintain, and hard to evolve. Additionally, such programming idioms rely on the explicit reference to class names to instantiate the different elements of a composed subsystem. Although programming patterns such as factories can be used to overcome this problem, they do so at the cost of adding an extra level of complexity to the code.

Despite of the absence of native support for dynamically manipulating the structure of programs, programmers sometimes take advantage of existing low-level reflection mechanisms to build systems that can adapt to their execution environment. As an example of this situation, consider a viewer that may select at run-time an appropriate third-party plug-in in order to handle different sorts of input data. Unfortunately, existing reflection mechanisms are not safe to build new structures although they are certainly useful to manipulate components in an untyped, and therefore not statically safe, way.

Middleware and component frameworks, such as Sun's Enterprise JavaBeans [96] and Microsoft's COM+ [32] provide infrastructures implementing base services which are typical of certain application scenarios. In this context, aggregation-based composition, supported by interconnection standards such as JavaBeans, COM and .NET, settled as the most common structuring mechanism. This preference reflects a shift of programming style from a pure, inheritance-based object-oriented style, towards the so-called "Component-Based Programming" idioms, which favour black-box composition [97]. Besides providing dynamic and late linking mechanisms, these standards also promote the usage of loosely-coupled webs of objects. Information hiding is here achieved by programming conventions, abstract interface descriptions, subscription based connections (event handling), and object factories, rather than by programming language mechanisms. Since the construction code for these object structures is not distinguished at the programming language level from any other code, the difficulties identified above, regarding the maintenance and evolution of systems, also persist. Static checking of architectural consistency (of the kind found, for example, in ML functor language or in mixin-based languages) is not performed at compile-time, and may cause hard to diagnose errors to show up only at run-time. The general problem seems to be that type safety for the code responsible for the system assemblage cannot be easily obtained from the static type safety of the underlying programming language.

Mixin and trait-based languages [16, 45, 15, 91, 80] do provide safe forms of loosely-coupled composition mechanisms for class-like modules. Although some of these core calculi allow the run-time construction of class structures, the industrial strength programming languages they inspire, e.g. Scala [76], although allowing some forms of dynamic object composition only deal with mixins and traits as compile-time values. Other ways of organising and combining classes in parameterised packages, based on the notion of mixin modules, were introduced in certain Java-like programming languages [74, 8]. Again, all package combinations are predefined at compile-time, and no context-aware system assembly may be safely expressed at the programming language level.

Despite being manually defined, the construction of object webs at run-time supports well an interesting notion of "just-in-time" construction of systems, which is not directly supported by any of the language-based approaches referred above. Moreover, the ad-hoc object web approach also allows for the dynamic evolution of structures by introducing new objects and refactoring the web connections at run-time. Again, these changes to the system structure are not defined separately from the remaining application code, and thus cannot be easily verified for static safety properties. The widespread use of mechanisms such as object serialisation, dynamic loading, and mobile code adds relevance to the general issue of finding expressive

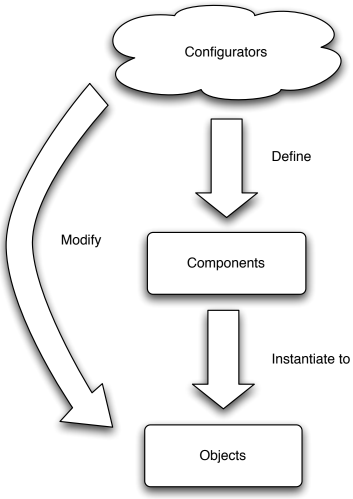

Figure 1.1: Model ingredients and interactions.

and safe programming language constructs to dynamically build and reconfigure systems by aggregation and replacement of components and objects.

1.2 A Model for Component-Oriented Programming

The main motivation of this work is therefore to propose and analyse supporting language-based mechanisms for the Component-Oriented Programming paradigm in the sense described above. We aim at answering the question: "How can programming idioms for structuring and maintaining programs, typical of the component-oriented programming style, be safely expressed at the programming language level?". We propose a programming model whose main ingredients and operations directly express component-oriented programming concepts. More concretely, we introduce a core object-oriented imperative language and extend it with new language constructs for expressing software composition and reconfiguration.

The main ingredients of our model, which are also first-class values in our programming language, are *objects*, *components*, and *configurators*. (see Figure 1.1.)

Objects are defined from components (cf. class instantiation). They aggregate state and functionality in the standard object-oriented sense and implement services which are specified by standard interface types. In opposition to objects in standard class-based programming languages, which tend to collapse a set of implemented services into one single interface, objects in our model provide explicitly separate views, distinguished by a name, on the services they implement. The different views of an object are available at access points called *ports*.

Components specify the structure and behaviour of objects by means of combination and adaptation, through scripting, of other components. At the component level ports also play an important role. A component declares a set of *required ports*, which denote external implementations of services that can be referred inside the component, and a set of *provided ports*, which export the services implemented by the component. The internal structure of components is specified by *configurators*. Any component establishes a visibility boundary allowing external access only to its provided and required ports. Additionally, the component border also forbids internal elements to refer to any name declared outside. Thus, in our model, component boundaries provide two-way information hiding. Not only this provides the usual concept of information hiding, from the outside towards the inside of the component, but it also provides a notion of information hiding from the inside of the component towards the outside. This makes components *closed values* which are usable independently of their evaluation context and indistinguishable from other components providing and requiring the same set of services. This equivalence relation is expressed at the level of typing and is extended to a richer, subsumption relation.

Configurators are first-class "functional-like" values that combine and adapt components by aggregation and scripting, in an implementation independent way. Our language defines a set of basic canonical configurators allowing the programmer to declare new required and provided ports, to introduce new components, to introduce new scripting blocks, or connect two existing elements. Configurators are composable in such a way that the resulting configurator produces the joint sequential effect of their parts. In opposition to components, that may only be used by connecting their ports, configurators can be combined with other configurators by referring to any elements they introduce or connect. (cf. "white-box" compatibility in Software Engineering terminology. We call *white-box* composition to the combination of configurators where all architectural elements are visible, in opposition to the hierarchical composition of components, where components include others but only the declared ports (required and provided) are visible and available for connection. The latter is called *black-box* composition.)

Thus, configurators are operations that describe the way in which components are aggregated and adapted; from the perspective of more usual compilation and linking process they can be compared (by analogy) to makefiles. Configurators can be incrementally built by composition of configurators of various sorts, and once a consistent network of elements is obtained, the configurator can then be "compiled" (the code in scripting blocks gets compiled in the context of composition) and "linked" (connections between ports are fixed) to produce a finished and executable entity, namely a component. Components can then be either used to produce objects in an application or used as elements of other compositions. Moreover, configurators can also specify changes in the structure of objects (cf. patch files). It is a distinctive feature of our model that both component assembly and object reconfiguration are specified uniformly using the same concept, the configurator.

The model we have informally described here is incrementally defined, in a precise way, in the chapters to follow. The programming language that expresses our model is defined by extending an imperative λ-calculus with composition constructs. The obtained language is closed under abstraction and application at a high-level of generality. Initially, we consider only the definition of configurators, to construct components and objects, then we extend the language to allow reconfiguration of objects, and finally we add subtyping and bounded parametric polymorphism. We follow the programming language design principles described in the next section, and before defining the language in a precise manner, we illustrate it by means of a few simple examples.

1.3 Design Principles

In this section, we present some principles that we assume as goals of our model and which guided the design of our programming language. The first principle states that all inconspicuous references between components must be avoided. Secondly, we aim at expressing in our model the dynamic construction of systems based on run-time decisions. As a complement to the dynamic construction of systems we also aim at expressing the evolution of objects. The last described principle concerns the typeful development of the model. We aim at providing programming language constructs with clear semantics and using a type language capable of capturing the essential properties of the ingredients and operations. We aim at statically ensuring the absence of run-time and load-time errors such as the lack of needed components, the call to unimplemented services, or the application of operations to incompatible object structures.

1.3.1 Explicit Functional Dependencies

The structural complexity of systems built using ad-hoc construction of object webs highly surpasses that of systems built using standard module mechanisms. But, the increased flexibility given by dynamically interconnecting system components has a negative effect on the perception one has about the dependencies of components.

In modern object-oriented platforms, the resolution of dependencies between system components is delayed to the moment of program loading which is usually performed using a lazy discipline. Since dependencies between modules are usually fixed, by means of names that directly refer to implementations, and are not explicitly declared in ways that the programmer is aware of them, severe problems may arise. Code breaks if a required component ceases to exist or the available implementation is different from the one known at compile-time. This sort of problems is more often found when components are used in contexts different to that of its definition. Errors of the sort “Method not implemented” or “Component not found” may happen in unexpected regions of the code where proper handling is not feasible.

We propose, as a fundamental concern in our model, that all functional dependencies are explicitly expressed and explicitly satisfied. We aim at defining loosely-coupled structures of components, based on object-oriented interface definitions and aggregation, as opposed to tightly-coupled inheritance-based hierarchies of classes or modules. Our approach to build such structures is to use explicit connection operations. The structures we obtain slightly resemble architectures of distributed systems defined in the style of classic Architecture Description Languages (ADLs) [49, 71]. In this kind of structures, the elements are viewed as black-boxes with publicly known connection points (input and output) and the connections between elements are “point-to-point”, possibly forming structures with mutually dependent components.

1.3.2 Dynamic Structuring Mechanisms

The ad-hoc construction of webs of objects not only allows for a flexible aggregation of parts of a system but also for the resulting structure to depend on information about the surrounding environment. This is not feasible in statically linked systems where all possible execution scenarios have to be considered in advance using a fixed structure of modules. To simulate this behaviour, the same monolithic program must predict and adapt to different run-time circumstances by explicit case analysis. Typically, the maintenance and evolution of this sort of code is a difficult task; it is mainly based on source editing in sparse locations of the code. Notwithstanding the already mentioned disadvantages of the ad-hoc aggregation of objects, the notion of dynamic construction of systems, referred above as “just-in-time”, seems to be unexplored at the level of programming language design. The assembly of systems at run-time, depending on run-time environment information, is a way of swiftly adapting and evolving an application to handle a large variety of situations, e.g. a web server that adapts its response to different browser programs, bandwidth conditions and even user preferences.

We believe that the definition of functionality and that of structure should be placed at the same level, in such a way that the effective structure of a program may depend on computational results. An adaptable program may start by obtaining information about its running environment and then configure itself in the most appropriate way. Although allowing the dynamic construction of components we fix the interface type information of the resulting components, thus avoid having dependent types in our language and the consequent phase distinction problems [22, 54].

1.3.3 Dynamic Reconfiguration of Objects

Software maintenance comprises two major sorts of tasks: error correction and upgrading of functionality. In general, these tasks require unanticipated changes to applications. The standard approach for evolving software, by editing the source code and producing new executables, inevitability forces the reinitialisation of target applications, which, in some circumstances, may cause major disruption in the provided services.

There is some work on evolution of module systems [90, 14, 93, 55, 37] but the native support at the programming language level for unanticipated modifications seems to be unexplored. Approaches to anticipated changes to implementation of objects are described by several authors [35, 89] but their motivation is to use reconfiguration as an additional abstraction mechanism rather than supporting the evolution of software. For instance, in Fickle [35], the concrete implementation of an object can vary within a set of classes that share a superclass. Each one of these subclasses represents a different "state" of the object, where different instance variables and different sets of methods are defined. More recently, [91] introduced a dynamic substitution mechanism for traits which allows to replace the implementation of a previously known set of methods.

One goal of our model is to support both the construction of components and the modification of objects at run-time. The use of language abstractions over composition operations allows for the definition and modification of components and objects to be uniformly expressed and controlled. This is achieved by defining small-grain composition operations and by mapping the structure of components to the network of elements and connections used to define objects. Such reconfigurations can be triggered in the course of the computation, and programs may even receive the reconfiguration scripts from an outer context, e.g. in case of an exception being raised by an object, the system may search for available correction scripts from a site of the original component developer, and in this way correct them.

1.3.4 Type Safety

In the historical development of programming languages, type systems revealed to be a fundamental tool in any programming language design by anticipating error detection to a phase prior to execution. Type systems ensure good properties of programs efficiently and without major user annotations. When modules are involved, a usual goal is to prove that their usage conforms with the available interface information and also that the code defining the module can be compiled separately, i.e. by ensuring that the declared interface is indeed implemented and that its code conforms to the interface of other modules.

The usage of sophisticated programming language constructs to abstract composition operations improves the expressiveness and readability of programs and also provides ground for static verification of semantic properties of objects, components and configurators. Our model ensures at compile-time that all objects, components, and configurators are well-formed with relation to their type information. In order to achieve this we describe our ingredients and in particular configurators with appropriate type information. While objects and components are typed with information that abstracts their implementation and solely describe their interfaces, configurators are typed so that their effects are fully described. Configurator types are defined in such a way that the compositionality of the operations over configurator values results in the compositionally of their types.

From the internal perspective of a component, type safety implies that all provided services are indeed implemented based only on the type information about the imported services. From the outside perspective, besides the correct usage of the provided services, the type system also verifies that a component is used only when all imported services are indeed available and connected.

This approach allows us to follow a type-based mechanism to ensure a notion of architectural soundness. In general we consider that components and objects are architecturally sound when all provided services are implemented and all dependencies between components are satisfied.

The model also ensures that all reconfigurations performed during the execution of a program produce well-formed objects and that posterior use of reconfigured objects is safe. However, an increased challenge arises from the interaction between the encapsulation mechanisms of components and objects and the operations needed to change their internal structure. The solution we found for this problem lies in a combination of dynamic and static typing that statically ensures that reconfiguration actions can be performed without run-time errors. We show that our model combines well with standard type abstraction mechanisms and subtyping. We add bounded parametric polymorphism and a rich subsumption relation for both component and configurator values.

Having stated these design principles we now informally describe the programming language where the concepts of our model are expressed.

1.4 The Programming Language

In this section, we illustrate the fundamental features of our model by means of two examples. In the first example, we implement the well-known Observer pattern [47]. We define basic components by composition and adaptation of other components, and define a factory method to work as a template for assembling the programming pattern from abstracted elements. We then use a second example, a word processing component, to illustrate how these features can be used to program software management operations such as automatic updating of code. The constructs of our language are used in the examples independently of the order used to present them in the dissertation; they are defined and explained in detail in the chapters to follow.

For the sake of legibility, we use some type abbreviations and a Java-like notation for variable and function declarations. Function types are written $(T)S$ for functions with a parameter of type T and return type S; interface types are collections of tagged function types; and the unitary type **void** is used with the usual meaning.

1.4.1 Representing the Observer Pattern

In the Observer pattern, there are two sorts of participating roles: there is one object that is the *subject* of observation and there is an undetermined number of *observer* objects. Each observer explicitly registers itself in the subject object to manifest its interest on receiving notifications on the occurrence of certain events (e.g. a state change). To that end, the subject keeps a list of the registered observer objects. The notification is performed by calling some predefined method on the observer objects and by passing a reference to the subject as argument (to allow further interaction).

Assuming that we want to observe a service of type ICollection defined as follows:

```
ICollection = {add:(int)void, remove:(int)void }
```

We specify the interface types of the objects in the observer pattern as follows:

```
IObserver = {update:(ICollection)void}
 ISubject = {register:(IObserver)void,
             unregister:(IObserver)void,
             notify:()void}
```

The behaviour of the participating objects is described as follows: an object that implements IObserver can be subscribed or unsubscribed from the interest of observing a subject by means of a call to the methods register or unregister, its method update is called as consequence of the method notify being called on the observed object.

Defining a generic subject We encode the functionality of the subject role in a single component, called CSubject, in the next fragment of code. (We assume given a generic list component, CList; it has a port named list whose methods are obvious from the example. First-class functions (λ-abstractions) are defined by expressions of the form **fun**(x:T){...}.)

```
c = (requires object:ICollection;
     provides subject:ISubject;
     uses L = CList;
     methods m {
       void register(o:IObserver) {
         L.list.add(o);
```

```
        },
        void unregister(o:IObserver) {
          L.list.remove(o);
        },
        void notify() {
          f = fun(o:IObserver){o.update(object);};
          L.list.iterate(f);
        }
      };
      plug m into subject);
  CSubject = compose (c);
```

The **compose** operation above builds a component whose structure is defined by a configurator passed as argument (c). Configurator c is the result of evaluating a composition operation, which in this case is a sequence of smaller composition operations. Composition operations can be either primitive, which add new elements to a configuration or connect two elements of a configuration, or they can denote the composition of two configurators $(-;-)$. The structure denoted by a compound configurator contains the elements of both operands and the connections they introduce may refer to elements on either side of the composition.

The primitive operations used to build the structure of CSubject, are described as follows. The operation **requires** introduces a required port object, which provides access to an imported service of type ICollection. The operation **provides** declares a provided port to export the subscription/notification functionality. This functionality is implemented in a method block introduced by the operation **methods** m{...}, where m is the local name denoting this block. The methods of m are made available at port subject by the **plug** operation. Notice that the methods of m rely on an existing component, CList, integrated in the structure of configurator c by the operation **uses** L =... where L is the local name by which it is referred to. It holds the references to all registered observers. Each composition operation introducing elements declares a local name, by which the new element is accessible, whose scope extends to the composition operations ahead. Notice, for instance, that the imported service (object) is directly passed to the observers in the call to method update.

Components are typed according to their required and provided service types and names. The internal elements declared by configurator c are encapsulated by the **compose** operation; only the provided and required ports remain visible in the type of the resulting component. In this particular case, the type of CSubject is of the form:

$$\{\mathsf{object}:\mathsf{ICollection}\} \Rightarrow \{\mathsf{subject}:\mathsf{ISubject}\}. \tag{1.1}$$

Component types comprise two sets of typed ports, one that enumerates the services upon which the component's implementation depends on (the ports on the left hand side), and another that indicates the services that the component implements (the ports on the right hand side). Notice that components with the same exact sets of provided and required ports are indistinguishable from the viewpoint of typing.

Implementing the pattern over a collection Now, consider that there is another component, CCollection, implementing a collection of integer values, typed by the component type T defined by:

$$T \triangleq \{\} \Rightarrow \{\mathsf{collection} : \mathsf{ICollection}\} \tag{1.2}$$

to which we would like to apply the Observer pattern. Notice that this component implements interface ICollection at a port named collection and that it does not need external services to do so (its set of required services is empty). Suppose that we want to observe the additions made to the collection. Then we may define a component that combines CSubject and CCollection as follows:

```
CObservableCollection = compose (
  provides collection:ICollection;
  provides subject:ISubject;
  uses c = CCollection;
  uses s = CSubject;
  methods m {
    void add(x:int) {
      c.collection.add(x);
      s.subject.notify();
    },
    void remove(x:int) {
      c.collection.remove(x);
    }
  };
  plug m into collection;
  plug s.subject into subject;
  plug c.collection into s.object);
```

Notice that this component preserves the original functionality of the collection component (at port collection) as well as the functionality associated with the subject role in the Observer pattern (at port subject). Both inner components, CSubject and CCollection are introduced in the structure of CObservableCollection via the composition operations **uses** s =... and **uses** c =.... They are referred by the local names s and c. The scope of these names extends to the composition operations ahead, which in this case are the method block declaration and the plug expressions. This means that each instance of component CObservableCollection will have one instance of component CCollection and an instance of component CSubject inside its structure. The two are then implicitly linked by means of scripting code (method block m) that watches the addition of elements to the collection and triggers a notification whenever it occurs (by calling method notify). The collection is provided to the inner component s, at port object, thus satisfying the functional dependency expressed in its type. The reference to a collection object resulting from this connection is passed on to the observer objects (in method notify of CSubject).

Defining Observers The observer objects can then be implemented as follows: (We assume given an interface type ILog with a method print accepting a string as argument.)

```
CCollectionObserver = compose (
  provides observer:IObserver;
  requires log:ILog;
  methods m {
    void update(c:ICollection){
      log.print("updated");
    }
  };
  plug m into observer)
```

In this case, whenever method update is called on port observer, a message is sent to the service connected to port log. Both components can then be instantiated and put to work as in the hypothetical sequence of instructions:

```
col = new CObservableCollection;
...
o = new CCollectionObserver with log := ...;
...
col.subject.register(o.observer);
...
col.collection.add(1);
...
```

The internal structure of object col is organised according to the structure of its generating component, CObservableCollection. The object has two visible provided ports connecting to method blocks inside its structure that implement the intended functionality. Recall the declaration of CObservableCollection and verify that, in this case, port collection leads directly to a method block of type ICollection, and port subject leads to a method block inside an instance of CSubject with type ISubject.

The instantiation of CCollectionObserver resorts to a more flexible form of instantiation. When components have unsatisfied required ports, an existing object can be connected to a component at instantiation time to implement them. This gives basic support for aliasing and sharing between objects. In this way different objects can share references to one common resource. The new object o receives a reference to an object implementing ILog in the **with** clause of the **new** expression. This mechanism can also be seen as a form of configuration of components. By providing different implementations to required ports of components at instantiation time, the programmer is influencing the behaviour of the resulting instances in different ways.

In this short sequence of instructions, we see that the observer object gets registered, by calling method register of the observable collection object col, and gets warned whenever some integer is added to the collection. This completes the definition of the Observer pattern using our composition-based language where the structure of the pattern is clearly expressed by aggregating two independent components. We now look in more detail to the typing of the composition operations presented here and we next generalise the assembly of the pattern by abstracting the role of the collection. Recall the composition operations used above: **provides**, to define a provided port; **requires**, to define a required port; **uses** to reuse a component in a composition; **methods** to define an implementation of a set of methods; **plug** to connect a service implementation to a port; and the sequencing of two composition operations to combine the effect of both.

Typing configurators The composition operations that build CObservableCollection yield first-class values which can be freely combined. We show here that a programming pattern such as the Observer pattern can be encoded more generically using a factory function. To explain this in a typeful way, we first need to describe how configurators are typed. Consider the composition of two configurators

(**provides** collection: ICollection ; **provides** subject:ISubject),

which lead the definition of CObservableCollection yields a configurator whose type is of the form

$$\{\} \Longrightarrow \{\mathsf{collection} \triangleright \mathsf{ICollection}, \mathsf{collection} \circ \mathsf{ICollection}, \mathsf{subject} \triangleright \mathsf{ISubject}, \mathsf{subject} \circ \mathsf{ISubject}\}. \qquad (1.3)$$

Configurator types consists of a pair of resource sets. *Resources* are bits of information representing the presence of certain conditions in partially built structures; they may be either required (the set on the left hand side of the long double arrow $\Longrightarrow$) or provided by a composition operation (the set on the right hand side). From the type expression above we know that the configurator transforms a component structure by adding a provided port collection with type ICollection and a provided port subject with type ISubject (the resources in the set on the right hand side with a $\triangleright$ symbol), it also notes that these ports are not yet implemented (by using the $\circ$ symbol).

Now, consider the operation (**uses** s = CSubject), that introduces an internal component into the structure of CObservableCollection, it is typed

$$\{\} \Longrightarrow \{\mathsf{s} \bullet \{\mathsf{collection} : \mathsf{ICollection}\}, \mathsf{s.subject} \bullet \mathsf{ICollection}\} \qquad (1.4)$$

which means that this operation introduces an internal element s, with type $\{\mathsf{collection} : \mathsf{ICollection}\}$, and that both s and its provided port s.subject are available for further composition operations (denoted by the $\bullet$ symbol).

The composition operation **plug** s.subject **into** subject, which should, in fact, be written with type annotations (**plug** s.subject:ISubject **into** subject:ISubject), has type:

$$U \triangleq \{\mathsf{s.subject} \bullet \mathsf{ISubject}, \mathsf{subject} \circ \mathsf{ISubject}\} \Longrightarrow \{\mathsf{s.subject} \bullet \mathsf{ISubject}\} \tag{1.5}$$

meaning that it satisfies the pending implementation of port subject (the resource with the ∘ symbol). A configurator typed by type U (1.5) "consumes" an open dependency (denoted by the resource subject ∘ ISubject) while maintaining the available resource used to satisfy it (the resource s.subject • ISubject). Notice that some of the composition operations used in these examples are written without type annotations, but they require them in the calculus definition ahead. This means only that these type annotations could be derived from the expressions or from the context of utilisation and are in the calculus for the sake of simplicity of the type system and type checking algorithm.

To complete the typing of primitive composition operations, consider the one that defines method block m,

```
methods m {
  void add(x:int) {
    c.collection.add(x);
    s.subject.notify();
  },
  void remove(x:int) {
    c.collection.remove(x);
  }
}
```

it has the following type S:

$$S \triangleq \{\mathsf{c} \bullet \{\mathsf{collection} : \mathsf{ICollection}\}, \mathsf{s} \bullet \{\mathsf{subject}:\mathsf{ISubject}\}\} \Longrightarrow \{\mathsf{m} \bullet \mathsf{ICollection}, \mathsf{c} \bullet \ldots, \mathsf{s} \bullet \ldots\}. \tag{1.6}$$

This type indicates that m depends on two inner components, denoted by the names c and s, i.e. that it has to be used in a context where c and s are available and have the types indicated by the resources, and that it introduces a new element m with interface type ICollection. Although not shown here, the expression introducing a method block is also decorated with extra information (as in the case of the **plug** operation). In this particular case the type decorations are essential to build the required resource set without resorting to sophisticated inference mechanisms, and thus help typing the method declarations.

Finally, the composition of two configurators yields a configurator whose type is a combination of the types of its parts. This combination propagates, to the resulting type, the required and provided resources of its parts and cancels the resources that denote connections between elements of both parts, e.g. the resource in type 1.3 noted ∘, that indicates that port subject is not yet implemented, is cancelled when combining such a configurator with a configurator of type *U* (1.5). The required resource set being empty means that the operation does not depend on any external element, i.e. it defines a network of elements which does not have a connection to any external element. In these conditions, and provided that it does not introduce unsatisfied resources, the configurator can be used to form a closed component value whose type is constructed just from the set of its required and provided ports.

Using Dynamic Composition Recall the definition of type *S* (1.6) of the composition operation **methods** m {...} above and *T* (1.2) which describes the interface of components such as CCollection. The function make defined next, takes any implementation of a collection (a component of type *T*) and a configurator with type S, that describes how the calls are intersected, and composes it according to the assembly of the Observer pattern shown here.

```
make = fun(C:T, M:S) {
  return compose (provides collection:ICollection;
                  provides subject:ISubject;
                  uses c = C;
                  uses s = CSubject;
                  M;
                  plug m into collection;
                  plug s.subject into subject;
                  plug c.collection into s.collection);
}
```

The function returns a component defined by a configurator which uses the function parameters (C and M) in its definition. The collection component denoted by the formal parameter C is introduced into the structure of the resulting component, as CCollection was before. Since the typing of the configurator is performed against the provided and required ports alone, it is possible to abstract the concrete implementation of the internal components of a composition and still validate the whole composition.

Additionally, we compose the formal parameter M in the configurator used to produce the component resulting from calling the function. Every time function make is called, the concrete elements of the argument instantiating parameter M are used to produce the resulting component. According to type *S* (1.6) of configurator M, the elements it introduces requires the presence of two inner components, named c and s, and introduces a new method block, named m. In the composition above, the occurrences of names c and s in the concrete implementations of M are bound to the elements introduced by the previous operations (**uses** c = C) and (**uses** s = CSubject). Furthermore, the method block it inserts (m) is made available to the plug operations that follow the composition of configurator M. So, the component resulting from calling make combines a given collection component C, with the known CSubject component, both adapted by a customly defined method block (introduced by the configurator M). Function make can then be used to extend any implementation of a collection typed by T with the Observer pattern as in:

```
addM = methods m {
          void add(x:int) {
            c.collection.add(x);
            s.subject.notify();
          },
          void remove(x:int) {
            c.collection.remove(x);
          }
        };
CObservableCollection = make(CCollection,addM);
```

The essence of programming patterns such as the Observer pattern can be captured in functions such as make above. Patterns which normally involve the assembly of webs of objects at run-time can be safely expressed in our model by means of factory functions where essential elements are abstracted. This allows for libraries of reusable patterns to be built and their correctness ensured prior to concrete utilisations, instead of relying on coding guidelines to implementing programming patterns.

Notice that the generic assembly of programming patterns illustrated here can benefit from the extension of this language with type abstraction (parametric polymorphism) and subtyping (inclusion polymorphism). For instance, by means of subsumption, function make above can accept any component whose type is a subtype of T. In our model, this means that these components provide at least the same services as the ones specified in T and require at most the required services of T (in this case there are none). Additionally, a type abstraction over the type of the collection can be used to generalise even more the usage of these patterns. (We assume abstract versions of the type definitions ISubject, T, S, and of component CSubject.)

```
make = All(X≤ICollection) fun(C:T<X>, M:S<X>) {
  return compose (provides collection:X;
                  provides subject:ISubject<X>;
                  uses c = C;
                  uses s = CSubject<X>;
                  M;
                  plug m into collection;
                  plug s.subject into subject;
                  plug c.collection into s.collection);
}
```

In this case the type of the ports collection and subject can be instantiated with any subtype of ICollection. The actual arguments of the function must then carry a component implementing the correct collection type and the method block that intercepts the methods of X.

Reconfiguration Remember that, in the example above, only the addition of elements is in fact logged by the subscribed observer objects (method add). If need arises to change the application to also log the removal of elements from a particular collection, which is an instance of CObservableCollection, and for some reason the system cannot easily be recompiled and restarted, then a safe reconfiguration mechanism that replaces pieces of objects while running is certainly useful. Consider the method block introduced by configurator M below, intercepting both calls of a CCollection component, and using a notification service of a CSubject component:

```
M = methods m {
        void add(x:int) {
          c.collection.add(x);
          s.subject.notify();
        },
        void remove(x:int) {
          c.collection.remove(x);
          s.subject.notify();
        }
    }
```

The type associated to configurator M is S (1.6), thus M can be used as argument to function make to produce new components implementing the observer pattern. Additionally, in our model, configurators can also be used to change the internal structure of objects. We define a language construct, **reconfig**, which applies a configurator to an object, thus introducing new elements and connections into its structure. Consider object col defined above as an instance of component CObservableCollection, it gets reconfigured in the following excerpt of code which first defines a configurator, using M defined above, and then reconfigures the object:

```
...
r = (M; plug m into collection);
...
reconfig x = r[col] in ... else ...
```

The configurator r defined above results from a composition of configurator M with a **plug** operation. It introduces a new method block, locally known by name m, and connects it to a port named collection. The **reconfig** expression applies configurator r to the actual structure of a target object, which in this case is object col. If we depict the inner structure of object col as an image of the structure of its generating component (CObservableCollection), it contains an object obtained from CCollection, an object obtained from CSubject, and a method block that adapts the functionality of both and is connected to port collection. The reconfiguration action, which applies configurator r to object col, changes the object in place, introduces a new method block into the structure of the object and replaces the implementation at port collection by the new functionality. The free occurrences of the names s and c in the expressions of the new method block m are bound to the elements already in the target object col. All existing references to object col are changed by aliasing. The expression **reconfig** declares a local name x which is binding in the **in** and **else** branches of the expression. In particular, x is typed in the **in** branch so that the all possible additions to the instance type (new provided ports) are visible.

Although configurator types reveal all the elements they introduce or connect, the static type information of the target objects only reveals the services they implement. Their implementation details are encapsulated. It is possible that, although providing the same set of ports, col is not an instance of CObservableCollection and its inner structure is incompatible with the modifications denoted by configurator r. In order to ensure the type safety of reconfiguration actions, we use a run-time compatibility test between the type of the configurator and the actual structure of target objects. Localised type information is stored in both object and configurator values to make this test possible. This allows for the actual modification of the object's structure to be performed only if there are guaranties that it will succeed. A successful reconfiguration is followed by the expression in the **in** branch. The **else** branch is executed otherwise. Type safety is ensured at compile-time by correlating the results of the run-time test with static properties.

This first example illustrates some of the basic features of the language. We now illustrate, in a second example, an interesting way in which these features can be used to implement self-updatable applications.

1.4.2 Implementing Automatic Updates

Applications that detect and apply updates automatically have emerged in the recent evolution of software systems. Although motivated for security reasons at first, this sort of mechanism rapidly generalised as an usual form of software maintenance and evolution in many non-critical applications. We show here that the kind of operations it requires can be expressed safely and explicitly in our model, without the need for restarting the applications.

Take the example of a word processor which may have, at the top level, a File Manager component that implements all the functionality related with persistence of data, autosave features, recovery of files, etc., a Text Manager component which implements basic text functionalities like insertion of text, searching, indexing, etc., a User Interface component which deals with the presentation layer, and a Proofing component which includes spell-checking and grammar analysers. Such a component can be defined as follows: (We omit elements, ports, and connections which are not directly related to the update mechanism we are illustrating.)

```
WordProcessor = compose (
  requires gw:IUpdateGateway;
  ...
  uses fm = FileMgr;
  uses tm = TextMgr;
  uses gui = WPGUI;
```

```
    uses prf = Proofing;
    ...
    methods m {
      void checkAll() {
        check(fm); check(tm);
        check(gui); check(prf);
      }

      void check(o:IUpdatableObject) {
        Key v = o.updateInfo.getKey();
        if(gw.availableScript(v))
          gw.getScript(v).update(o);
      }
    }
  )
```

To implement a self update mechanism, component WordProcessor depends on an external service that provides *update scripts* for objects given an identification key. This service is specified by the interface IUpdateGateway as follows:

```
IUpdateGateway = { availableScript:(Key)boolean, getScript:(Key)IUpdateScript }
```

Both methods are based on the key value of a particular object, method availableScript indicates if an update is available and method getScript fetches an update script associated with the given key. Type Key is of no particular relevance and will remain opaque for the rest of the presentation. The values that method getScript returns are of type IUpdateScript. Type IUpdateScript declares a method update and is defined as follows:

```
IUpdateScript = { update:(IUpdatableObject)void }
```

where IUpdatableObject is an object type with a single port updateInfo with a method getKey,

```
IUpdatableObject = { updateInfo: { getKey:()Key }}
```

Consider that all the types of inner elements of component WordProcessor, visible in the definition above, have the common supertype IUpdatableObject. This means that, besides their own functionality, they all provide a unique identification key on a port updateInfo.

Our word processing application watches for update releases on a set of its inner elements. It interacts with the directory of update scripts, available at port gw, that maps key values to components capable of reconfiguring objects that have the same key. When requested, the instances of these components are retrieved and used in client applications to modify their inner elements. This process is triggered by a call to method checkAll which selectively calls method check using one object as argument at a time. Method check works as follows: it gets the key value of the object by calling getKey at its port updateInfo, accesses the directory service to find an update script for that particular key by calling method available at port gw, and, if there is an update available, fetches an update script of type IUpdateScript, by calling getScript at port gw, and calls its method update on our target object to make the reconfiguration. A typical implementation of an update script can be:

```
Script = compose (
  provides p:IUpdateScript;
  methods m {
    r = (uses x = X; plug x.q into p),
    void update(o:IUpdatableObject) { reconfig r[o]; }
  };
  plug m into p);
```

Notice that the instances of Script encapsulate primitive reconfiguration operations; they carry a configurator, in the local variable r of method block m, which defines the modifications to be made in the target object. In this case, configurator r introduces a new inner element (an instance of component X) and connects its port q to a port of the reconfigured object (p), thus hiding the former implementation of p. Notice that the value of component Script includes all the necessary component values referred in r to reconfigure the target object. The reconfiguration is performed when the method update is called by the client component (in method check to an instance of Script); r is applied to the target object o provided that there is no mismatch between the type of the configurator and the actual structure of the object.

Our type system ensures that there is no disruption due to reconfiguration actions, even in the presence of separate compilation. The run-time test, that guards the reconfiguration action, ensures that target objects have all the internal elements which are necessary to perform the changes declared by the configurator value. From the result of this test we can decide whether to apply the configurator to the object, knowing that it will perform all the changes without any run-time error, or to not apply it at all, thus avoiding possible run-time errors due to mismatches between the operations and the structure of the object. A well-typed reconfiguration action can therefore be considered as atomic, it terminates in all cases and causes no run-time errors. Hence, there is no need for performing dynamic tests during the modification of the object and there is no need to maintain any "undo" information to be used in case of error. This property is particularly important in software management operation such as the one illustrated in this last example. If systems are to be assembled by using third-party components, then the update scripts issued for upgrading or correcting objects are more likely to be developed and compiled in the development context of original component (where their internal structure is known). Although the structure of these update scripts is unknown at compile-time, they can be safely accepted and used in the context of a client application.

Furthermore, notice that these update scripts can be dynamically composed to meet the needs for a particular object. In the example above, a key value representing the available information about the implementation of an object is passed onto a generic service that manages update scripts (typed IUpdateGateway). This information can be taken into account to dynamically produce an update script that brings each particular object up-to-date. For instance, consecutive versions of update scripts can be composed to obtain a single one, or, depending on the software licensing a client has, different elements can be introduced into its objects.

The two examples just presented here illustrate the flexibility of our programming model to represent the dynamic construction and maintenance of component-based systems in a typeful way. We now enumerate the contributions of our work and describe the structure of the remainder dissertation.

1.5 Evolution and Contributions

In this section we briefly describe the sequence of advances that lead to the present stage of our work and enumerate its main contributions.

An initial proposal of our model, presented in [81], describes a programming language capturing the essential ingredients of object-oriented component programming styles, such as explicit context dependence, subtype polymorphism at the level of both components and objects, late composition, and avoidance of inheritance in favour of composition. A type system was defined, with types assigned to (first-class) components and objects, ensuring run-time safety of compositions. However, although in such a model components may be dynamically composed, the structure of objects gets permanently fixed at instantiation time, thus excluding any possibility of dynamic reconfiguration.

Our model evolved from a monolithic composition sublanguage to a more flexible one, presented in [88]. This is basically the language presented throughout this dissertation. We present a core component-oriented programming language with a minimal set of architectural primitives which yield configurator values that can be freely manipulated and composed to build sophisticated structures, we also define the modification of objects using the exact same set of composition operations. Moreover, we develop a type system that statically enforces, besides the absence of the usual run-time errors, the consistency of component compositions and the atomicity of dynamic reconfiguration.

Some intermediate results were described in [85, 83, 84], and a prototype compiler for a practical language that integrates the fundamental concepts of our model, componentJ, was also designed and implemented [86].

We proposed, in [87], a definition for a subtyping relation in a class-based language using the subtyping discipline of kernel-Fun [27] with equi-recursive types, which significantly improves other approaches [30, 31]. Although the motivation of this work was to find a rich subtyping relation between component types, this work stands out as an independent result which we use to extend our model. This result is here presented in the more general plain kernel-Fun calculus.

The contributions of our work can therefore be summarised as follows:

- We define a programming model for software composition expressive enough to model many sophisticated software management operations, typical of component-based systems, involving dynamic composition, configuration and reconfiguration. We provide type safe modularisation mechanisms for component-based software development, which abstract common programming idioms such as the ad-hoc construction of object webs.
- We provide programming language constructs for expressing the reconfiguration of objects. The preservation of the hierarchical structure of components at the level of objects allows to uniformly express both the construction of components and the reconfiguration of objects.
- We instantiate our model on a core programming language, where the structures of components and the reconfiguration of objects are type safe. We define a type system that captures structural as well as computational safety properties of the language and combines, in a single type safety result, properties of progress, type preservation, and structural soundness of configurators, components, and objects.
- We integrate our component-based programming model with language features such as structural subtyping, type recursion, and bounded parametric polymorphism. On doing this, we tackle the challenging problem of subtyping second-order equi-recursive types and develop new interesting techniques. We uniformly extend an approximation to first-order equi-recursive types with explicit management of the binding of type variables and a similarity relation on subtyping judgements. As a result we obtain a simple coinductive definition of the subtyping relation and a corresponding coinductive, syntax-directed, subtyping algorithm.
- We provide a prototype compiler of a Java-like component language which was a first attempt to integrate the composition-based mechanisms defined in our model in a mainstream object-oriented programming language.

We next describe the structure of the dissertation.

1.6 Structure of the Dissertation

The remainder of this dissertation is structured to gradually present precise definitions for the concepts informally addressed in this introductory chapter. At the end of each chapter, we make some remarks about the subjects we think are relevant and relate the presented work with that of other authors in dedicated sections. The contents of the remaining chapters are as follows:

- Chapter 2 defines the base language of our model, an imperative λ-calculus with mutable records. This chapter introduces notation and techniques that will be used throughout the dissertation.
- Chapter 3 presents our basic component-based language. It extends our base language with composition operations and the corresponding type system. We prove a type safety result, and as consequence we conclude that all well-typed component expressions produce well-structured instances and that no run-time errors occur due to lack of implemented services.
- Chapter 4 extends the component language presented in Chapter 3 with run-time reconfiguration of objects. We extend our language in order to apply composition operations to the structure of objects, and in this way change their internal structure. We also extend the type safety result so that the well-formedness property of components and instances is preserved in reconfigured instances.
- In Chapter 5 we present some fundamental results on subtyping recursive second-order types. We start by introducing the existing approaches to subtyping of first-order recursive types which we then extend to define a new algorithm for second-order types by uniformly extending them. We use standard coinductive techniques to show the termination and correctness of our algorithm and prove correctness results.
- We apply the results of Chapter 5 to our component language and present, in Chapter 6, a new type system and subtyping relation that allows flexible reuse of component values. We define a subtyping relation on component and configurator types which satisfies the principle of safe substitution. We also present a complete typing algorithm for the component-based language.
- We then draw some conclusions about the proposed programming model and describe some future directions in Chapter 7.

Finally, towards the end of this dissertation there are two appendixes: Appendix A provides complementary technical details of the proofs in this dissertation, and Appendix B presents the componentJ language. We illustrate the features of our model that were implemented in our prototype compiler by means of small examples. We roughly sketch some basic tools and gadgets that allow the integration of componentJ with either Java client programs or native Java components.

Chapter 2

Preliminaries

In this chapter, we introduce basic concepts and techniques that will be used throughout the dissertation. More concretely, we define an untyped imperative λ-calculus with mutable records, λ_R, and then its typed version, λ_R^τ. The operational semantics of this calculi are formalised using a big step operational semantics, and type safety properties are proved using standard syntax-directed techniques. Our aim is also to lay down the technical foundations upon which the formalisation of our component-oriented programming model will be developed.

2.1 λ_R — An Untyped λ-calculus with Mutable Records

In this section we define λ_R, an untyped λ-calculus with mutable records, which is the base language for the component language presented in subsequent sections. We start by defining the syntax and semantics of the language, but first we need to define notation about basic elements such as variables, record labels, and memory locations to support the next definitions.

Basic notation

Let $\mathcal{L}$ be a set of denumerable labels denoted by symbols ℓ, ℓ', ℓ_i; let $\mathcal{V}$ be a set of denumerable variables denoted by x, y, z; and let $\mathtt{Loc}$ be a set of denumerable memory locations noted by the symbols $\imath$, $\imath'$, $\imath_i$.

We now define our core λ_R language. It includes standard abstraction and application, record construction, selection, and assignment expressions.

Syntax

Definition 2.1 (Terms). *The language λ_R is defined by the abstract syntax in Figure 2.1.*

$$
\begin{array}{lll}
e \; ::= & & \lambda_R \text{ terms} \\
\mid & x & \text{variable} \\
\mid & \lambda x.e & \text{abstraction} \\
\mid & e(e) & \text{application} \\
\mid & \{\ell_i = e_i \;^{i \in 1..n}\} & \text{record} \\
\mid & e.\ell & \text{selection} \\
\mid & e.\ell := e & \text{assignment} \\
\mid & l & \text{location} \\
\mid & \mathsf{nil} & \text{null value}
\end{array}
$$

Figure 2.1: Abstract syntax of λ_R.

$$
\begin{array}{lll}
v \; ::= & & \text{values} \\
\mid & \lambda x.e & \text{abstraction} \\
\mid & \{\ell_i = l_i \;^{i \in 1..n}\} & \text{record} \\
\mid & l & \text{location} \\
\mid & \mathsf{nil} & \text{null value}
\end{array}
$$

Figure 2.2: Abstract syntax of λ_R values.

In an abstraction $\lambda x.e$, variable x is binding in expression e, the record expression associates labels to expressions in a record value, the selection expression projects a record value to the field with the given label, and assignment modifies this value. As usual, we consider terms up-to renaming of bound variables (α-equivalence).

Although defined as such, we do not consider, in practise, locations (l) to be part of the source language, they should be regarded only as run-time values introduced during evaluation by a specialised allocation operation.

Among these expression forms we define those that define the possible results of evaluating a λ_R term as follows::

Definition 2.2 (Values). *The set of values $\mathcal{U}_R \subseteq \lambda_R$ is defined by the abstract syntax in Figure 2.2.*

Notice that record values are those whose fields are already evaluated and stored in the heap.

We use the standard notation $FV(e)$ to denote the free variables of an expression e, and $FL(e)$ to denote the set of locations occurring in e.

Definition 2.3 (Free variables)**.** *We define the set $FV(e)$ of free variables of expression e as follows:*

$$\begin{aligned}
FV(x) &\triangleq \{x\}\\
FV(\lambda x.e_1) &\triangleq FV(e_1)\backslash\{x\}\\
FV(e_1(e_2)) &\triangleq FV(e_1) \cup FV(e_2)\\
FV(\{\ell_i = e_i\ ^{i\in 1..n}\}) &\triangleq \textstyle\bigcup_{i\in 1..n} FV(e_i)\\
FV(e_1.\ell) &\triangleq FV(e_1)\\
FV(e_1.\ell := e_2) &\triangleq FV(e_1) \cup FV(e_2)\\
FV(\imath) &\triangleq \emptyset\\
FV(\mathsf{nil}) &\triangleq \emptyset.
\end{aligned}$$

Definition 2.4 (Occurrences of locations)**.** *We define the set $FL(e)$ of location occurrences in expression e as follows:*

$$\begin{aligned}
FL(x) &\triangleq \emptyset\\
FL(\lambda x.e_1) &\triangleq FL(e_1)\\
FL(e_1(e_2)) &\triangleq FL(e_1) \cup FL(e_2)\\
FL(\{\ell_i = e_i\ ^{i\in 1..n}\}) &\triangleq \textstyle\bigcup_{i\in 1..n} FL(e_i)\\
FL(e_1.\ell) &\triangleq FL(e_1)\\
FL(e_1.\ell := e_2) &\triangleq FL(e_1) \cup FL(e_2)\\
FL(\imath) &\triangleq \imath\\
FL(\mathsf{nil}) &\triangleq \emptyset.
\end{aligned}$$

We now define capture avoiding substitution on terms in the expected way.

Definition 2.5 (Substitution)**.** *A substitution (θ) is a finite mapping from variables and labels to expressions.*

We denote by $[x \leftarrow e]$ (respectively $[\ell \leftarrow e]$) the singleton substitution that maps x (respectively ℓ) to e. Notice that we will not use substitution of labels until section 3.1. We write $\mathrm{Dom}(\theta)$ to denote the domain of the substitution θ and define the codomain of a substitution θ by $\mathrm{Img}(\theta) \triangleq \bigcup\{FV(\theta(x)) \mid x \in \mathrm{Dom}(\theta)\}$. We write $\theta\backslash\{x\}$ to restrict the domain of the substitution θ by eliminating the substitution of x.

Definition 2.6 (Application of substitution)**.** *The application of a substitution θ to an expression e, written $e\theta$ is defined in Figure 2.3*

Semantics

The semantics of λ_R is defined with relation to a memory heap representing a global execution state. Hereafter, we define notions on heaps, locations, and expressions for using in forthcoming language definitions. We define heaps as follows:

$$\begin{aligned}
x\theta &\triangleq \theta(x) where\ \ x \in \mathrm{Dom}(\theta)\\
y\theta &\triangleq y \quad where\ \ y \notin \mathrm{Dom}(\theta)\\
(\lambda x.e)\theta &\triangleq \lambda x.(e\theta') \quad where\ \ \theta' = \theta\backslash\{x\}\\
(e_1(e_2))\theta &\triangleq (e_1\theta)(e_2\theta)\\
\{\ell_i = e_i\ {}^{i\in 1..n}\}\theta &\triangleq \{\ell_i = e_i\theta\ {}^{i\in 1..n}\}\\
(e_1.\ell)\theta &\triangleq (e_1\theta).\ell\\
(e_1.\ell := e_2)\theta &\triangleq (e_1\theta).\ell := (e_2\theta)\\
\imath\theta &\triangleq \imath\\
\mathsf{nil}\theta &\triangleq \mathsf{nil}
\end{aligned}$$

Figure 2.3: Application of subtitutions to terms.

Definition 2.7 (Heap). *A* heap *S is an assignment of values to locations. The heap that assigns v_i to $\imath_i$ with $i \in 1..n$ is written $\{\,\imath_i{\mapsto}v_i\ {}^{i\in 1..n}\}$, the empty heap is written $\emptyset$.*

We use the following notations for operations on heaps: $S(\,\imath)$ to denote the value associated with $\imath$ in S, $S[\,\imath \mapsto v]$ to denote a heap S updated with a new assignment, and $\mathrm{Dom}(S)$ to denote the domain set of S. We use $\mathrm{new}(S)$ to denote a fresh memory location in S, and write $\mathsf{nil}(S)$ to denote the set of locations in S that map to nil. Notice that locations are also values and therefore chains of locations may occur in a heap: we use $\mathrm{deref}_S(\,\imath)$ to denote the last location of the chain starting in $\imath$. The precise definition of these basic operations is given by:

Definition 2.8 (Operations on heaps). *Consider the heap S, the locations $\imath, \imath_1, \ldots, \imath_n$, and the values $v, v_1, \ldots, v_n$. We have:*

- $\mathrm{Dom}(\{\,\imath_i{\mapsto}v_i\ {}^{i\in 1..n}\}) \triangleq \{\,\imath_i\ {}^{i\in 1..n}\}$.
- *if $S = \{\ldots, \imath \mapsto v, \ldots\}$ then $S(\,\imath) \triangleq v$ otherwise $S(\,\imath)$ is undefined.*
- *if $S = \{\,\imath_1{\mapsto}v_1, \ldots, \imath_n{\mapsto}v_n\}$, and $\imath \notin \mathrm{Dom}(S)$*
  then $S[\,\imath \mapsto v] \triangleq \{\,\imath_1{\mapsto}v_1, \ldots, \imath_n{\mapsto}v_n, \imath \mapsto v\}$.
- *if $S = \{\,\imath_1{\mapsto}v_1, \ldots, \imath_n{\mapsto}v_n\}$, and $\imath = \imath_j$ for some $j \in 1..n$*
  then $S[\,\imath \mapsto v] \triangleq \{\,\imath_1 \mapsto v_1, \ldots, \imath_j \mapsto v, \ldots, \imath_n \mapsto v_n\}$.
- $\mathrm{new}(S) \triangleq \imath$ *where* $\imath \in \mathtt{Loc}\backslash\, \mathrm{Dom}(S)$.
- $\mathsf{nil}(S) \triangleq \{\,\imath \mid S(\,\imath) = \mathsf{nil}\}$.
- $\imath_i\ {}^{i\in 1..n} \in \mathrm{Dom}(S)$ *is a S-*chain *if $S(\,\imath_i) = \imath_{i+1}\ \ \forall_{i\in 1..n-1}$*
- $\mathrm{deref}_S(\,\imath_1) \triangleq \imath_n$ *when $\exists\,\imath_i\ {}^{i\in 1..n} \in \mathrm{Dom}(S)$ such that $\imath_i\ {}^{i\in 1..n}$ is a S-chain and $S(\,\imath_n) \notin \mathtt{Loc}$.*
  $\mathrm{deref}_S(\,\imath)$ *is undefined otherwise.*

The usual notations on sequences apply to these operations, for instance $S[\,l_i{\mapsto}v_i\ ^{i\in 1..n}]$ denotes the sequence of updates $S[\,l_1 \mapsto v_1] \ldots [\,l_n \mapsto v_n]$.

Notice that locations can lead to other locations and therefore cyclic chains of locations may potentially exist in heaps, leading from a location to itself after a number of indirections (intuitively, $\mathrm{deref}_S(\,l)$ is undefined). We now introduce some terminology on locations which allows us to characterise them more precisely. We say that a location participating in a cycle is *undefined*; we characterise as *direct* a location that leads to a value which is not a location in one dereferencing step; in a chain of locations we say that l *leads-to* l' if there are a number of indirections in the heap leading from one to the other (possibly zero); if at the end of a chain of location there is a value which is not a location then we say that the location *refers-to* the value.

Definition 2.9 (Notation about locations)**.**

- (Direct location) *A location* l *is* *direct* *when* $S(\,l) \notin \mathsf{Loc}$ *and* $S(\,l) \neq \mathsf{nil}$.
- (Leads to) *A location* l *leads-to* l' *in* S *when* $\mathrm{deref}_S(\,l) = l'$.
- (Refers to) *A location* l *refers-to* *a value* v *in* S *when* l *leads-to* l' *in* S *and* $S(\,l') = v$.
- (Undefined) *A location is* *undefined* *when it* *refers-to* *no value, i.e. it starts a cyclic chain.*

Some of these situations do not arise when evaluating a λ_R term with relation to an empty heap but are introduced here for the sake of generality in languages presented ahead.

Operational semantics

We now define the semantics of λ_R by means of a big-step operational semantics, following a call-by-value evaluation strategy. The semantics is based on judgements of the form

$$e; S \downarrow v; S'$$

Such a judgement asserts that the expression e evaluated with relation to heap S yields the value v and modifies the heap S to obtain S'. Before introducing the evaluation relation of λ_R, we establish some useful terminology.

(Eval Value)

$$v; S \downarrow v; S$$

(Eval Application)

$$\frac{e_1; S \downarrow \lambda x.e; S' \quad e_2; S' \downarrow v; S'' \quad e[x \leftarrow v]; S'' \downarrow v'; S'''}{e_1(e_2); S \downarrow v'; S'''}$$

(Eval Select)

$$\frac{\begin{array}{c} e; S \downarrow \iota; S' \quad \iota' = \mathrm{deref}_S(\iota) \\ S'(\iota') = \{\ldots, \ell = \iota'', \ldots\} \end{array}}{e.\ell; S \downarrow S'(\iota''); S'}$$

(Eval Assign)

$$\frac{\begin{array}{c} e_1; S \downarrow \iota; S' \quad \iota' = \mathrm{deref}_{S'}(\iota) \\ S'(\iota') = \{\ldots, \ell = \iota'', \ldots\} \\ e_2; S' \downarrow v; S'' \end{array}}{e_1.\ell := e_2; S \downarrow v; S''[\iota'' \mapsto v]}$$

Figure 2.4: Evaluation rules for λ_R.

Definition 2.10 (Notation)**.**

- (Closed Heap) *We say that a heap S is* closed *if all locations occurring in the values in S are elements of* $\mathrm{Dom}(S)$.

  S is closed $\triangleq \ \forall \iota \in \mathrm{Dom}(S).\ \forall \iota' \in FL(S(\iota)).\ \iota' \in \mathrm{Dom}(S).$

- (Closed Expression) *Given an expression e and a closed heap S, we say that e is* closed *in S if all locations occurring in e are elements of* $\mathrm{Dom}(S)$.

  e is closed in S $\triangleq FL(e) \subseteq \mathrm{Dom}(S)$.

- (Valid Configuration) *If an expression e is closed in a heap S we then say that the pair $(e; S)$ is a* valid configuration.

  $(e; S)$ is a valid configuration $\triangleq$ *e is closed in S.*

The evaluation rules shown in Figure 2.4 define inductively the evaluation relation of λ_R as follows:

Definition 2.11 (Evaluation)**.** *Let $e \in \lambda_R$ and a heap S such that $(e; S)$ is a valid configuration. The evaluation relation of an expression e to a value v, written $e; S \downarrow v; S'$ is defined inductively by the rules in Figure 2.4.*

We use the standard evaluation rules for values and application: Rule (Eval Value) and Rule (Eval Application). Notice that, following the usual call-by-value evaluation strategy, abstractions are considered atomic until applied to an argument. Side effects on partial evaluations are represented in the updates of heaps S', S'', and S''' with relation to heap S. Notice that the evaluation of records, Rule (Eval Record), builds a new record in the heap, containing locations where field values get stored. Retrieval and modification of field values are performed in the heap by Rules (Eval Assign) and (Eval Select). We disallow the direct manipulation of locations by syntactically imposing the assignment form $e.\ell := e$ and by implicitly dereferencing locations to values when selecting a field (Algol-like state instead of ML-like state). Hence, in this particular operational semantics, all locations resulting from evaluating a record expression are *direct*, leading to records in a single dereferencing step. Unless there are chains in the initial evaluation heap, it is not possible for a λ_R term to create non-trivial chains in the resulting heap, i.e. with length greater than 2. For the sake of later developments in this work, i.e. in languages using the λ_R calculus as their base language, we design selection and assignment operations to be "chain-transparent" by means of the operation $\mathrm{deref}_{\_}(-)$. This means that, on evaluating a selection or assignment expression, the record at the end of the resulting chain of locations is selected and, in the case of assignment, gets modified (see Rules (Eval Select) and (Eval Assign) in Figure 2.4). Other than this implicit dereferencing behaviour, the semantics presented here follows standard lines.

Example 2.12. We now illustrate the semantics of λ_R, by means of a small example. Consider the following application expression: $(\lambda x.(x.\ell := \{\}))(\{\ell = \mathsf{nil}\})$

The derivation of the evaluation judgement is as follows:

(Eval Value)	$(\lambda x.(x.\ell := \{\}));\emptyset \downarrow (\lambda x.(x.\ell := \{\}));\emptyset$
(Eval Value)	$\mathsf{nil};\emptyset \downarrow \mathsf{nil};\emptyset$
(Eval Record)	$(\{\ell = \mathsf{nil}\});\emptyset \downarrow l_0;S'$ where $S' = \{l_o \mapsto \{\ell = l_1\}, l_1 \mapsto \mathsf{nil}\}$
(Eval Value)	$l_0;S' \downarrow l_0;S'$ where $l_0 = \mathrm{deref}_S(l_0)$ and $S'(l_0) = \{\ell = l_1\}$
(Eval Record)	$\{\};S' \downarrow l_2;S''$ where $S'' = \{l_o \mapsto \{\ell = l_1\}, l_1 \mapsto \mathsf{nil}, l_2 \mapsto \{\}\}$
(Eval Assign)	$(l_0.\ell := \{\};S' \downarrow l_2;S'''$ where $S''' = \{l_o \mapsto \{\ell = l_1\}, l_1 \mapsto l_2, l_2 \mapsto \{\}\}$
(Eval Application)	$((\lambda x.(x.\ell := \{\}))(\{\ell = \mathsf{nil}\}));\emptyset \downarrow l_2;S'''$

Notice that the records are evaluated to locations in the heap (l_0 and l_2). These locations lead to record values whose fields are also locations (l_1). l_1 then leads to the corresponding field value of the record. The assignment expression is evaluated by changing the contents of these locations in the heap.

Given this untyped language, we now define its typed version and associated type system. We next prove type safety of the language.

2.2 λ_R^τ — A Typed λ-calculus with Mutable Records

We now present a typed version of λ_R and a type system to ensure type safety of the language, that is to say that "well-typed programs do not go wrong". We use this introductory exer-

$$\begin{array}{lll}
\tau ::= & & \text{types} \\
\mid & \tau \rightarrow \tau & \text{function type} \\
\mid & \{\!| \ell_i : \tau_i \,^{i \in 1..n} |\!\} & \text{record type}
\end{array}$$

Figure 2.5: Abstract syntax of λ_R^τ types.

$$\begin{array}{lll}
e ::= & & \lambda_R \text{ terms} \\
\mid & x & \text{variable} \\
\mid & \lambda x : \tau . e & \text{abstraction} \\
\mid & e(e) & \text{application} \\
\mid & \{ \ell_i = e_i \,^{i \in 1..n} \} & \text{record} \\
\mid & e.\ell & \text{selection} \\
\mid & e.\ell := e & \text{assignment} \\
\mid & \iota & \text{location value} \\
\mid & \mathsf{nil} & \text{null value}
\end{array}$$

Figure 2.6: Abstract syntax of λ_R^τ.

cise to modularly explain some basic techniques used throughout the dissertation and avoid cluttering our presentation of several concepts with basic definitions and technical details.

Types

We start by introducing a language of type expressions following the standard definition for the typed λ-calculus with records [25, 78].

Definition 2.13 (Types). *The types $\mathcal{T}_R$ of λ_R are defined by the abstract syntax in Figure 2.5.*

Syntax

We now use the type language $\mathcal{T}_R$ to define an explicitly typed version of the λ_R language, called λ_R^τ.

Definition 2.14 (Terms). *The language λ_R^τ is defined by the abstract syntax in Figure 2.6.*

Notice that the only modification with relation to λ_R is the explicit typing of formal parameters in abstractions. Values are changed accordingly:

Definition 2.15 (Values). *The set of values $\mathcal{U}_R^\tau \subseteq \lambda_R^\tau$ is defined by the abstract syntax in Figure 2.7.*

We now describe the semantics of this new language, λ_R^τ.

$$\begin{array}{lcll} v & ::= & & \text{values} \\ & \mid & \lambda x:\tau.e & \text{abstraction} \\ & \mid & \{\ell_i = \iota_i \ ^{i\in 1..n}\} & \text{record} \\ & \mid & \iota & \text{location value} \\ & \mid & \mathsf{nil} & \text{null value} \end{array}$$

Figure 2.7: Abstract syntax of λ_R^τ values.

(Eval Value)

$$v;S \downarrow v;S$$

(Eval Application)$^\tau$

$$\frac{e_1;S \downarrow \lambda x:\tau.e;S' \quad e_2;S' \downarrow v;S'' \quad e[x \leftarrow v];S'' \downarrow v';S'''}{e_1(e_2);S \downarrow v';S'''}$$

(Eval Select)

$$\frac{\begin{array}{c} e;S \downarrow \iota;S' \quad \iota' = \mathrm{deref}_{S'}(\iota) \\ S'(\iota') = \{\ldots,\ell = \iota'',\ldots\} \end{array}}{e.\ell;S \downarrow S(\iota'');S'}$$

(Eval Assign)

$$\frac{\begin{array}{c} e_1;S \downarrow \iota;S' \quad \iota' = \mathrm{deref}_{S'}(\iota) \\ S'(\iota') = \{\ldots,\ell = \iota'',\ldots\} \\ e_2;S' \downarrow v;S'' \end{array}}{e_1.\ell := e_2;S \downarrow v;S''[\iota'' \mapsto v]}$$

Figure 2.8: Evaluation rules for λ_R^τ.

Operational semantics

For the sake of completeness we present the complete rule system in Figure 2.8. The operational semantics of λ_R^τ is defined as follows:

Definition 2.16 (Evaluation). *Let $e \in \lambda_R^\tau$ and a heap S such that $(e;S)$ is a valid configuration. The evaluation relation of an expression e to a value v, written $e;S \downarrow v;S'$ is defined inductively by the rules in Figure 2.8.*

It is clear that the operational semantics of λ_R^τ is the same as the one of λ_R, just a minor change in the abstract syntax was introduced. Type annotations are needed here to simplify the implementation of the type system defined next. The type system is designed to ensure that all evaluations of well-typed expressions are free of run-time errors.

Type system

The typing of λ_R^τ expressions is formally defined by a typing judgement of the form $\Delta \vdash e : \tau$ where e is an expression in λ_R^τ, τ is a type expression in $\mathcal{T}_R$ and Δ is a typing environment.

$$\begin{array}{lcll}
\Delta, \Gamma & ::= & & \text{typing environments} \\
 & \mid & \phi & \text{empty environment} \\
 & \mid & \Delta, x : \tau & \text{type assignment to a variable} \\
 & \mid & \Delta, \iota : \tau & \text{type assignment to a location}
\end{array}$$

Figure 2.9: Abstract syntax of typing environments.

$$\text{(Env Empty)} \quad \phi \vdash \diamond \qquad \text{(Env Var)} \quad \frac{\Delta \vdash \tau\ \mathsf{ok} \quad x \notin \mathrm{Dom}(\Delta)}{\Delta, x : \tau \vdash \diamond} \qquad \text{(Env Loc)} \quad \frac{\Delta \vdash \tau\ \mathsf{ok} \quad \iota \notin \mathrm{Dom}(\Delta)}{\Delta, \iota : \tau \vdash \diamond}$$

$$\text{(Type Fun)} \quad \frac{\Delta \vdash \tau\ \mathsf{ok} \quad \Delta \vdash \sigma\ \mathsf{ok}}{\Delta \vdash \tau \rightarrow \sigma\ \mathsf{ok}} \qquad \text{(Type Record)} \quad \frac{\Delta \vdash \tau_i\ \mathsf{ok} \quad \forall_{i \in 1..n} \qquad \Delta \vdash \diamond}{\Delta \vdash \{\!| \ell_i : \tau_i{}^{i \in 1..n} |\!\}}$$

Figure 2.10: Validation rules for typing environments.

Definition 2.17 (Typing Environment)**.** *For $x \in \mathcal{V}$, $\iota \in \mathtt{Loc}$, and $\tau \in \mathcal{T}_R$ the set $\mathcal{D}$ of all typing environments is defined by the abstract syntax in Figure 2.9.*

We write $\Delta \vdash e : \tau$ to assert that expression e has type τ with relation to environment Δ. Our typing environments assign types not only to variables but also to locations. This is used later on to define a notion of heap typing, needed in the subject reduction (type preservation under evaluation) proof. With respect to typing environments, we write $\mathrm{Dom}(\Delta)$ to denote the declared variables and locations in Δ, and define a notion of valid typing environment as follows:

Definition 2.18 (Valid Typing Environment)**.** *A typing environment Δ is* valid *if the judgement $\Delta \vdash \diamond$ is derivable by the rules in Figure 2.10.*

The validation of typing environments is defined together with a notion of well-formed type expressions. Here, validity basically ensures that all types used correctly build on basic types (the empty record type), or use valid type expressions on that typing environment.

We now define the valid type judgements of λ_R^τ by means of a rule system.

Definition 2.19 (Typing relation)**.** *The judgement $\Delta \vdash e : \tau$ is valid if it is derivable by the rules in Figure 2.11.*

Notice that the nil value is not allowed by typing to appear in expressions.

This rule system provides a type for any (typable) expression with relation to a typing environment. Variables and locations are typed according to information in the typing environment, abstractions are typed with function types relating their parameter and result types, the typing

(Val Var)
$$\frac{x : \tau \in \Delta}{\Delta \vdash x : \tau}$$

(Val Location)
$$\frac{l : \tau \in \Delta}{\Delta \vdash l : \tau}$$

(Val Abstraction)
$$\frac{\Delta, x : \tau \vdash e : \sigma}{\Delta \vdash \lambda x : \tau . e : \tau \rightarrow \sigma}$$

(Val Application)
$$\frac{\Delta \vdash e_1 : \tau \rightarrow \sigma \quad \Delta \vdash e_2 : \tau}{\Delta \vdash e_1(e_2) : \sigma}$$

(Val Record)
$$\frac{\Delta \vdash e_i : \tau_i \quad \forall i \in 1..n}{\Delta \vdash \{\ell_i = e_i{}^{i \in 1..n}\} : \{\!|\ell_i : \tau_i{}^{i \in 1..n}|\!\}}$$

(Val Select)
$$\frac{\Delta \vdash e : \{\!|\ldots, \ell : \tau, \ldots|\!\}}{\Delta \vdash e.\ell : \tau}$$

(Val Assign)
$$\frac{\Delta \vdash e_1 : \{\!|\ldots, \ell : \tau, \ldots|\!\} \quad \Delta \vdash e_2 : \tau}{\Delta \vdash e_1.\ell := e_2 : \tau}$$

Figure 2.11: Typing rules for λ_R^τ.

of applications uses a function type, checks the compatibility of the argument type and types the result accordingly. Records are typed by associating each label to the type of its initialising expression. Selection and assignment check for the presence of a value with type record and the correct label. Assignment expressions are also checked for the compatibility between the type of the record field and the type of the value assigned to it.

Together with the semantics defined before, we now enunciate and prove type safety of the λ_R^τ language. Type safety implies that the result of evaluating an expression always produces a value of the expected type, and that the evaluation of well-typed expressions does not get stuck due to run-time errors.

2.2.1 Type Safety

We show here the property of subject reduction, meaning that the result of evaluating an expression always yields a value of the expected type. To that end, we extend the operational semantics with rules to explicitly capture all situations where it is not possible to soundly apply an evaluation step. These (error trapping) rules yield a distinguished value wrong, and always halt the evaluation process. As a consequence, we conclude that run-time errors do not occur during the evaluation of well-typed expressions.

A run-time error is defined to occur whenever an operation is undefined, this includes conditions such as: application using a value which is not an abstraction, assignment to a value which is not a location, selection of a field on a value which is not a record or does not possess the relevant label (notice that this case includes calling a method on a nil reference), and so on. Essentially, we include all situations in which the operational semantics of Figure 2.8 gets stuck (yields a finitely failed derivation). To conclude the reasoning we show that the evaluation derivations of well-typed expressions do not contain error-trapping rules.

Thus, we next define:

(Wrong Call)

$$\frac{e_1; S \downarrow v; S' \quad v \neq \lambda x : \tau . e}{e_1(e_2); S \downarrow \mathsf{wrong}; S'}$$

(Wrong Select)

$$\frac{e; S \downarrow v; S' \quad v \notin \mathsf{Loc}}{e.\ell; S \downarrow \mathsf{wrong}; S}$$

(Wrong Select 2)

$$\frac{e; S \downarrow \iota; S' \quad S'(\mathrm{deref}_{S'}(\iota)) \neq \{\ldots, \ell = \iota', \ldots\}}{e.\ell; S \downarrow \mathsf{wrong}; S}$$

(Wrong Assign)

$$\frac{e_1; S \downarrow v; S' \quad v \notin \mathsf{Loc}}{e_1.\ell := e_2; S \downarrow \mathsf{wrong}; S}$$

(Wrong Assign 2)

$$\frac{e_1; S \downarrow \iota; S' \quad S'(\mathrm{deref}_{S'}(\iota)) \neq \{\ldots, \ell = \iota', \ldots\}}{e_1.\ell := e_2; S \downarrow \mathsf{wrong}; S}$$

Figure 2.12: Error trapping rules for λ_R^τ.

Definition 2.20 (Extended Evaluation). *Let $e \in \lambda_R^\tau$ and a heap S such that $(e; S)$ is a valid configuration. The evaluation relation of an expression e to a value v, written $e; S \downarrow v; S'$ is defined inductively by the rules in Figures 2.8, and 2.12.*

We say that a heap is well-typed with relation to a typing environment if the values referred by its locations are either nil or have the expected type. We define the typing of heaps as follows:

Definition 2.21 (Typing of Heaps). *For any typing environment Γ and heap S, we say that Γ* types *S if* $\mathrm{Dom}(\Gamma) \cap Loc \subseteq \mathrm{Dom}(S)$ *and* $\forall \iota \in \mathrm{Dom}(S)$ *we have that $\Gamma \vdash \iota : \tau$ and*

1. *ι is `undefined`,*

2. *ι `refers-to` nil, or*

3. *ι `refers-to` v and $\Gamma \vdash v : \tau$.*

We now state our first subject reduction theorem, that captures local invariants of evaluation judgements with relation to types. These local invariants rely on global typing information about the heap (kept in the context of the proof). In the proof of subject reduction below we use two standard properties of any typing relation: weakening and preservation of types under substitution.

Lemma 2.22 (Weakening). *For all typing environments Δ, Δ', all expressions $x, e \in \lambda_R^\tau$, and $\tau \in \mathcal{T}_R$: If $\Delta, \Delta' \vdash e : \tau$ and $x \notin Dom(\Delta, \Delta')$ then $\Delta, x : \tau', \Delta' \vdash e : \tau$.*

Proof. By induction on the height of the derivations and in the case of the last rule used. □

Lemma 2.23 (Substitution). *For all typing environments Δ, Δ', all expressions $x, e \in \lambda_R^\tau$, and $\tau \in \mathcal{T}_R$: If $\Delta, x : \tau, \Delta' \vdash e : \tau'$ and $\Delta \vdash v : \tau$ then $\Delta, \Delta' \vdash e[x \leftarrow v] : \tau'$.*

Proof. By induction on the height of the derivations and in the case of the last rule used. □

Now, by restricting the use of both locations and nil as source expressions we enunciate subject reduction as follows:

Theorem 2.24 (Subject Reduction). *Let $(e; S)$ be a valid configuration in $\lambda_R^\tau \backslash \{\mathsf{nil}\}$ and let Γ be a typing environment typing S and $\mathsf{nil}(S) = \emptyset$:*
If $\Gamma \vdash e : \tau$ and $e; S \downarrow v; S'$ then:

a) there is a Γ' that extends Γ and types S',

b) $\Gamma' \vdash v : \tau$,

c) v is either an abstraction, or a location that is either `undefined` or `refers-to` a record, and

d) $\mathsf{nil}(S') = \emptyset$.

Proof Sketch. We prove this theorem by induction on the length of the evaluation derivation and in the cases of the last rule used in the evaluation. We show that these rules are never applicable in the cases which evaluate to wrong. (for the complete proof see appendix A on page 173). □

Notice that, from Theorem 2.24 c), we conclude that no wrong values can be generated as results of evaluating well-typed expressions.

This concludes our presentation of the base language λ_R^τ. We are now ready to present our core component calculus.

Chapter 3

A Core Component Calculus

In this chapter, we introduce the component calculus λ_χ. The λ_χ calculus is obtained by extending the basic language λ_R, defined in Chapter 2, with language constructs aiming to capture typical idioms of the Component-Oriented programming paradigm and the design principles put forward in the Introduction.

We develop a type system that statically enforces, besides the absence of more usual run-time errors, consistency of component compositions. The design of λ_χ is semantically motivated by considering a domain of *configurators*, *components*, and *objects*; all such entities are first-class in our model. Intuitively, configurators correspond (by analogy) to the usual notion of "makefile". Essentially, each configurator contains a series of instructions (architectural primitives) about how to assemble a component. Thus, language expressions that evaluate to configurator values may be seen as counterparts of configuration scripts, the kind of programs used in software configuration management systems to dynamically generate makefiles. Configurators which do not refer to external entities may generate components, by means of a **compose** primitive. Components are linked pieces of code (*cf.*, a class or a module), that may be further composed with other components and scripting code, in configuration scripts, or instantiated, by means of a **new** primitive, to yield objects. Methods can then be called on the appropriate ports of an object, in order to invoke its services.

Section 3.1 defines an untyped component calculus λ_χ, and Section 3.2 defines its typed version, λ_χ^τ. We then define a type system that ensures a property of type safety of λ_χ^τ expressions which implies architectural soundness of configurators, components, and objects. Thus, we ensure that the evaluation of well-typed λ_χ^τ expressions, which may involve both basic and compositional computations, do not cause any run-time errors.

3.1 λ_χ — An Untyped Component Calculus

In this section, we define an untyped core programming language with a small set of primitive mechanisms, which we propose as the fundamental ones to support object-oriented component programming. More concretely, we extend the language λ_R of Section 3.1 with constructs that capture the ingredients and operations of our model already discussed in the Introduction.

We define composition operations in the language which evaluate to configurator values: to introduce new required and provided ports, to introduce new internal components and method blocks, to connect two ports, or to compose other configurators. We also introduce two special operations, one that produces components from configurators (corresponding roughly to a linking operation) and another that produces objects from components (corresponding roughly to an instantiation operation as found in class-based object-oriented languages). We use ground composition operations in configurator and component values to represent these operations at run-time, and use the basic values of λ_R, abstractions, mutable records, and locations, to encode the resulting objects.

The syntax and semantics of the our next language, named λ_χ, are then defined as follows:

Syntax

Definition 3.1 (Terms)**.** *The language λ_χ is defined by the abstract syntax in Figure 3.1.*

As may be seen in the previous definition, the first seven constructs of λ_χ are the expressions of λ_R, so that λ_R is a proper fragment of λ_χ (in fact we will later conclude that λ_χ is a conservative extension of λ_R). We now intuitively explain the expression forms that are added to λ_R. The compose expression (*component creation*) and the instantiation expression new (*instantiation*), and a set of primitive composition operations, namely the requires expression, the provides expression, the component introduction expression $x[-]$, the method block expression $x_{\{-\}}[- = -]$, the plug expression, and the configurator composition expression $-;-$. We also define the run-time representations of configurator and component values as language terms, respectively $\mathsf{conf}(-)$ and $\mathsf{comp}(-)$. In addition to locations and nil values, as in λ_R, the values of the form $\mathsf{conf}(-)$ and $\mathsf{comp}(-)$ are not expected to occur in source programs, typically they occur only as a result of evaluating expressions.

We now describe the intuitive semantics of composition expressions.

Each composition expression may be seen as a primitive configuration script, represented at run-time by a configurator. Configurators are stateless values intended to produce a specific structural effect on a component structure. As explained in Chapter 1, general composition operations may be primitive or obtained by composition with $(-;-)$ from two other composition operations. Primitive composition operations may introduce new elements in an component structure, adapt (through scripting code represented by method blocks), or simply interconnect

$$
\begin{array}{lcll}
e & ::= & & \lambda_\chi \text{ terms} \\
 & | & x & \text{variable} \\
 & | & \lambda x.e & \text{abstraction} \\
 & | & e(e) & \text{application} \\
 & | & \{\ell_i = e_i \ {}^{i\in 1..n}\} & \text{record} \\
 & | & e.\ell & \text{selection} \\
 & | & e.\ell := e & \text{assignment} \\
 & | & \ell & \text{port label} \\
 & | & \mathsf{compose}\ e & \text{component creation} \\
 & | & \mathsf{new}\ e\ \mathsf{with}\ \ell_j := e_j\ {}^{j\in 1..m} & \text{instantiation} \\
 & | & \mathsf{requires}\ \ell & \text{required port} \\
 & | & \mathsf{provides}\ \ell & \text{provided port} \\
 & | & x[e] & \text{component introduction} \\
 & | & x_L[\ell_i = \lambda x_i.e_i\ {}^{i\in 1..n}] & \text{method block} \\
 & | & \mathsf{plug}\ \pi\ \mathsf{into}\ \pi & \text{plug} \\
 & | & e;e & \text{configurator composition} \\
 & | & \imath & \text{location} \\
 & | & \mathsf{nil} & \text{null value} \\
 & | & \mathsf{conf}(e) & \text{configurator} \\
 & | & \mathsf{comp}(e) & \text{component} \\
 & & & \\
\pi & ::= & \ell \mid x \mid x.\ell & \text{port name}
\end{array}
$$

In the (method block) expression, L denotes a finite set of labels.

Figure 3.1: Abstract syntax of λ_χ.

existing elements. In a *composition* $(e_1; e_2)$, we let any element of the structure introduced by the configurator yield by e_1 to be referred and connected to elements introduced by the configurator yield by e_2, thus producing more elaborate component structures. We may say that this kind of composition is *"white-box"*, reusing software engineering terminology. The primitive operations which are basic for defining composition are the following:

- The expression $\mathsf{requires}\ \ell$ declares a named port in the current composition context to import a service from the external context;

- The expression $\mathsf{provides}\ \ell$ declares a named port to export a service to the external context;

- The expression $x_L[\ell_i = \lambda x_i.e_i\ {}^{i\in 1..n}]$, introduces a basic scripting block containing method implementations. Such a method block is referred by the name x which is binding in the expressions e_i, and local to the composition context. The elements of the surrounding component structure, which are named in the set of labels L, may be referred in the bodies of these abstractions;

- The component introduction expression $y[e]$ introduces an internal component in the composition context. Such a component results from evaluating e. Again, such an element may be referred in the composition context by the name y, which is local to the composition context. The provided and required ports of the introduced component become available through the name y.

- The expression plug is used to establish connections between elements. The various elements introduced by a configurator can be interconnected by plug π_1 into π_2 expressions. Such a plug expression declares that method invocations at port π_2 should be redirected to port π_1. Plug sources (π_1) can be either required ports of the current composition context (ℓ), representing imported functionality, provided ports of components in the same composition context ($x.\ell$), or locally defined method blocks (x).

  On the other hand, the targets of these connections (π_2) can be either provided ports of the current composition context (ℓ) that get "implemented", or required ports of internal components ($x.\ell$), whose requirements get "satisfied".

As motivated in the Introduction, a completed composition context establishes a visibility boundary allowing external access only to its provided and required ports. Additionally, the component border also forbids internal elements to refer to any name declared outside, thus providing two-way information hiding.

The evaluation of composition operations, discussed above, produce configurator values. Configurators are the key elements of the expressions compose and new, described next.

Given an expression e denoting a configurator, compose e yields a component value which "freezes" the configurator's structure inside a component value in such a way that it can only be further composed using its provided and required ports as connection points, by means of a composition operation $x[-]$. In more usual terms, the compose operation can be interpreted as a process that links modules to produce other modules, possibly changing representations, although in our model this operation is captured abstractly. Notice that the compose operation only makes sense if applied to configurators whose architecture is consistent in a precise sense, i.e. where all elements are correctly interconnected, if all provided ports have implementations assigned to them, and if all requirements of internal components are satisfied by some internal connection. This and other properties are ensured by the type system defined below. The result of a compose operation is always a component value.

Component values can be instantiated, with the new expression, to yield objects. Objects must have all their required ports linked to compatible implementations. Such open dependencies may get satisfied by means of assignments in the with clause of the new expression,

$$\begin{array}{lcll}
v & ::= & & \text{values} \\
 & | & \lambda x.e & \text{abstraction} \\
 & | & \{\ell_i = l_i\ {}^{i \in 1..n}\} & \text{record} \\
 & | & l & \text{location} \\
 & | & \mathsf{nil} & \text{null value} \\
 & | & \mathsf{conf}(c) & \text{configurator} \\
 & | & \mathsf{comp}(c) & \text{component} \\
 & & & \\
c & ::= & & \text{ground composition operations} \\
 & | & \mathsf{requires}\ \ell & \text{required port} \\
 & | & \mathsf{provides}\ \ell & \text{provided port} \\
 & | & x[v] & \text{component introduction} \\
 & | & x_L[\ell_i = \lambda x.e_i\ {}^{i \in 1..n}] & \text{method block} \\
 & | & \mathsf{plug}\ \pi\ \mathsf{into}\ \pi & \text{plug} \\
 & | & c; c & \text{composition} \\
 & & & \\
\pi & ::= & \ell \mid x \mid x.\ell & \text{port name}
\end{array}$$

Figure 3.2: Abstract syntax of λ_χ values.

hereafter referred as *plug-assignments*. In this way, it is possible to complete and configure existing components at instantiation time by linking the newly created object with others already present in the system. These features seem fundamental to define an imperative language with dynamic construction and manipulation of components, with aliasing and sharing of stateful objects. In a degenerate setting, with construction of new compositions is the only mechanism to satisfy required ports, the border of a component would have to be extended (by composition) until it includes all needed functionality. However, such a scenario will probably turn out unrealistic in a resource based model like the ones found in most object-oriented languages, where the state of the underlying platform is shared between all objects (for instance, for the purpose of input/output).

From the above discussion, it becomes clear that composition expressions have a double role. On one hand, in a computational context, they evaluate to stateless configurator values and represent first-class configuration scripts. On the other hand, when used in a composition context, they are applied in order to cause effects on a ongoing component composition. (in Chapter 4 our model will be extended so that configurators are also used to reconfigure component instances.)

After having defined and discussed the language of λ_χ we single out the values that may result from evaluation of expressions:

Definition 3.2 (Values). *The set of values $\mathcal{U}_\chi \subseteq \lambda_\chi$ is defined by the abstract syntax in Figure 3.2.*

Notice that, besides the values of λ_R, the results of λ_χ include components and configurators, of the form $\mathsf{comp}(c)$ and $\mathsf{conf}(c)$ where c is a fully evaluated composition operation, referred here as a *ground composition operation*. Objects are defined using record values.

A configurator value, which is the run-time representation of a composition operation, is represented in our model simply by the corresponding composition operation itself. An actual implementation may choose a more adequate or useful representation for configurators. On the other hand, the representation of a component value is represented by enclosing with the constructor comp the instructions for the construction of component instances. Although syntactically similar, configurator values and component values actually denote quite different semantic entities. In the case of a component value the internal composition operation c represents the structure of a finished (fully linked) component. In the case of a configurator value the internal composition operation c represents a mapping from component structures to component structures, thus being more a functional value than a first-order value (see Chapter 1). In particular, a configurator value does not in general define a complete (instantiable) component structure.

Both components and configurators are intended to be pure values in the sense that they do not possess active state. This means that both components and configurators may be freely copied and possibly transmitted on communication channels in a possible extension of this model with concurrency and distribution. Although in the untyped calculus λ_χ this property is not necessarily ensured, that will be the case in the typed version developed in the next section. Objects, on the other hand, are stateful entities constructed as specified by their generating component. Constructor $\mathsf{comp}(-)$ is an annotation that distinguishes a component structure as complete, which can only be composed without accessing its internal elements (in $x[-]$ operations) and can also be used to produce instances (in new expressions). Configurators, on the other hand, typically denote incomplete structures that are to be composed with others in a flat structure (by $-;-$ operations), by referring their internal elements (using their local names).

In the language λ_χ, we define labels as language expressions to allow the use of port names in expressions, more concretely, in method blocks. Thus, we need to use the notion of substitution (Definition 2.5) of labels by expressions.

We extend the application of a substitution to expressions as follows:

Definition 3.3 (Application of substitution)**.** *The application of a substitution θ to an expression e of λ_χ, written $e\theta$, is defined in Figure 3.3.*

Notice that application of a substitution to the field initialiser expressions of method blocks is restricted on the names that are intended to be be interpreted only in the composition context as discussed in Section 3.1.1 below. For the sake of simplicity, we also use the notation $e[r]$ where r is a record $\{\ell_i = e_i\ ^{i\in 1..n}\}$ to denote the substitution $[\ell_i \leftarrow e_i\ ^{i\in 1..n}]$.

We are now ready to define the operational semantics of λ_χ.

$$
\begin{aligned}
x\theta &\triangleq \theta(x)\\
y\theta &\triangleq y \quad \textit{where} \;\; y \notin \mathrm{Dom}(\theta)\\
(\lambda x.e)\theta &\triangleq \lambda x.(e\theta') \quad \textit{where} \;\; \theta' = \theta\backslash\{x\}\\
(e_1(e_2))\theta &\triangleq (e_1\theta)(e_2\theta)\\
\{\ell_i = e_i \;^{i\in 1..n}\}\theta &\triangleq \{\ell_i = e_i\theta \;^{i\in 1..n}\}\\
(e_1.\ell)\theta &\triangleq (e_1\theta).\ell\\
(e_1.\ell := e_2)\theta &\triangleq (e_1\theta).\ell := (e_2\theta)\\
\ell\theta &\triangleq \theta(\ell)\\
\ell\theta &\triangleq \ell \quad \textit{where} \;\; \ell \notin \mathrm{Dom}(\theta)\\
(\mathsf{compose}\;\; e)\theta &\triangleq \mathsf{compose}\;(e\theta)\\
(\mathsf{new}\; e\; \mathsf{with}\; \ell_j := e_j \;^{j\in 1..m})\theta &\triangleq \mathsf{new}\; e\theta\; \mathsf{with}\; \ell_j := e_i\theta \;^{j\in 1..m}\\
(\mathsf{requires}\; \ell)\theta &\triangleq \mathsf{requires}\; \ell\\
(\mathsf{provides}\; \ell)\theta &\triangleq \mathsf{provides}\; \ell\\
(x[e])\theta &\triangleq x[e\theta]\\
x_L[\ell_i = \lambda x_i.e_i \;^{i\in 1..n}]\theta &\triangleq x_L[\ell_i = (\lambda x_i.e_i)\theta' \;^{i\in 1..n}] \;\; \textit{where}\; L = \ell'_i \;^{i\in 1..m} \;\textit{and}\; \theta' = \theta\backslash\{x, \ell'_i \;^{i\in 1..m}\}\\
(\mathsf{plug}\; \pi_1\; \mathsf{into}\; \pi_2)\theta &\triangleq \mathsf{plug}\; \pi_1\; \mathsf{into}\; \pi_2\\
(e_1; e_2)\theta &\triangleq e_1\theta; e_2\theta\\
\imath\theta &\triangleq \imath\\
\mathsf{nil}\theta &\triangleq \mathsf{nil}\\
(\mathsf{conf}(e))\theta &\triangleq \mathsf{conf}(e\theta)\\
(\mathsf{comp}(e))\theta &\triangleq \mathsf{comp}(e\theta)
\end{aligned}
$$

Figure 3.3: Application of substitutions to terms.

Operational semantics

The operational semantics of λ_χ is defined by a big-step evaluation semantics that maps λ_χ expressions to λ_χ values. The semantics is based on two evaluation judgements, a judgement to compute the value of computational expressions and a judgement to compute the effect of composition operations.

More precisely, we write

$$e; S \downarrow v; S'$$

to mean that expression e when evaluated in heap S yields a value v and heap S'. We also write

$$s; c; S \Downarrow s'; S'$$

to mean that composition operation c is applied to a partially built object s with relation to a heap S and yields the partially built object s' with respect to heap S'. In such a judgement we refer the partially built object s as the *composition context*. Thus, the effects of composition

(Eval Requires)

$$\mathsf{requires}\ \ell; S \downarrow \mathsf{conf}(\mathsf{requires}\ \ell); S$$

(Eval Provides)

$$\mathsf{provides}\ \ell; S \downarrow \mathsf{conf}(\mathsf{provides}\ \ell); S$$

(Eval Plug)

$$\mathsf{plug}\ \pi_1\ \mathsf{into}\ \pi_2; S \downarrow\ \mathsf{conf}(\mathsf{plug}\ \pi_1\ \mathsf{into}\ \pi_2); S$$

(Eval Sequence)

$$\frac{e_1; S \downarrow \mathsf{conf}(c_1); S' \qquad e_2; S \downarrow \mathsf{conf}(c_2); S'}{(e_1; e_2); S \downarrow \mathsf{conf}(c_1; c_2); S'}$$

(Eval Uses)

$$\frac{e; S \downarrow v; S'}{x[e]; S \downarrow \mathsf{conf}(x[v]); S'}$$

(Eval Method Block)

$$x_L[\ell_i = \lambda x.e_i\ ^{i \in 1..n}]; S \downarrow\ \mathsf{conf}(x_L[\ell_i = \lambda x.e_i\ ^{i \in 1..n}]); S$$

(Eval Compose)

$$\frac{e; S \downarrow \mathsf{conf}(c); S'}{\mathsf{compose}\ e; S \downarrow \mathsf{comp}(c); S'}$$

(Eval New) $(s = (r, e, p), \quad r = \{\ell_i \mapsto \imath_i\ ^{i \in 1..n}\}, \quad \imath = \mathrm{new}(S))$

$$\frac{e; S \downarrow \mathsf{comp}(c); S_0 \quad \mathbf{0}; c; S_n \Downarrow s; S_{n+1} \quad e_i; S_{i-1} \downarrow v_i; S_i\ \forall_{i \in 1..n}}{\mathsf{new}\ e\ \mathsf{with}\ \ell_i := e_i\ ^{i \in 1..n}; S \downarrow \imath; S_{n+1}[\imath \mapsto s][\imath_i \mapsto v_i\ ^{i \in 1..n}]}$$

Figure 3.4: Evaluation rules for λ_χ.

operations are characterised in the semantics by incremental modifications of a composition context.

The semantics is defined by the rule system in Figures 2.4, 3.4, and 3.5 as follows:

Definition 3.4 (Evaluation)**.** *Let $e \in \lambda_\chi$ and S be heap such that $(e; S)$ is a valid configuration. The evaluation relation of an expression e to a value v, written $e; S \downarrow v; S'$ is defined inductively by the rules in Figure 2.4, 3.4, and 3.5.*

We now prove that our evaluation relation is well-defined by proving that all evaluation judgements produce well-formed heaps, well-formed values, and that the resulting values are closed in the resulting heaps.

Lemma 3.5.

1. *For all expressions e and heaps S that form a valid configuration $(e; S)$, if $e; S \downarrow v; S'$ then v is a value, S' is a heap, and $(v; S')$ is a valid configuration.*

2. *For all records s, expressions c, and heaps S that form valid configurations $(s; S)$ and $(c; S)$, If $s; c; S \Downarrow s'; S'$ for some expression c, then s' is a record, S' is a heap, and $(s'; S')$ is a valid configuration.*

Proof. By mutual induction on the height of the derivations of the judgements $e; S \downarrow v; S'$ and $s; c; S \Downarrow s'; S'$ and by case analysis on the last rule used. □

Notice that we refer to the rules of the base language, λ_R (Figure 2.4), and introduce new rules for the remaining expressions (Figures 3.4 and 3.5). The judgement form $e; S \downarrow v; S'$,

(App Sequence)

$$\frac{s; c_1; S \Downarrow s'; S' \quad s'; c_2; S' \Downarrow s''; S''}{s; (c_1; c_2); S \Downarrow s''; S''}$$

(App Requires) $(\iota = \text{new}(S))$

$$(r, e, p); (\mathsf{requires}\ \ell); S \Downarrow (r \oplus \{\ell = \iota\}, e, p); S[\iota \mapsto \mathsf{nil}]$$

(App Provides) $(\iota = \text{new}(S))$

$$(r, e, p); (\mathsf{provides}\ \ell); S \Downarrow (r, e, p \oplus \{\ell = \iota\}); S[\iota \mapsto \mathsf{nil}]$$

(App Uses)

$$\frac{\mathsf{new}\ v; S \downarrow \iota; S'}{(r, e, p); x[v]; S \Downarrow (r, e \oplus \{x = \iota\}, p); S'}$$

(App Method Block) $(\iota, \iota_i\ {}^{i \in 1..n} = \text{new}(S), \quad v_i' = v_i[(r, e, p)][x \leftarrow \iota])$

$$(r, e, p); x_L[\ell_i = v_i\ {}^{i \in 1..n}]; S \Downarrow (r, e \oplus \{x = \iota\}, p); S[\iota \mapsto \{\ell_i = \iota_i\ {}^{i \in 1..n}\}][\iota_i \mapsto v_i'\ {}^{i \in 1..n}]$$

(App Plug)

$$s; \mathsf{plug}\ \pi_1\ \mathsf{into}\ \pi_2; S \Downarrow s; S[\text{select}_S(s, \pi_2) \mapsto \text{select}_S(s, \pi_1)]$$

Figure 3.5: Application rules for composition operations in λ_χ.

defining the evaluation of an expression e with relation to a memory heap S, depends on the judgement form $s; c; S \Downarrow s'; S'$, in rule (Eval New), to define the application of a composition operation c to a composition context. We use **0** to denote an empty object. A composition context is a partially built instance s where the effects of the operations are accumulated. In the end, this composition context is taken as the resulting instance.

The basic composition operations provides, requires, and plug, are interpreted as themselves as described above, and thus directly stored in configurator values since they are closed expressions, Rules (Eval Requires), (Eval Provides), (Eval Plug). The fields of method blocks (which are abstractions) are also, by definition, values of the language. So, they may be also directly stored in configurator values, as indicated in Rule (Eval Method Block).

In Rule (Eval Uses), the case of the introduction of an internal component ($x[e]$), the resulting value depends on the evaluation of the expression e to a component value v. Then, the ground composition operation ($x[v]$) gets stored in the resulting configurator value.

Finally, evaluating a composition, in Rule (Eval Sequence), produces a configurator containing the composition of the two configurators resulting from evaluating its two operands.

The evaluation of a compose e expression (Rule (Eval Compose)), evaluates expression e to yield a configurator value. This value is expected to be well-formed, but it might be not, because the configurator may not ensure the construction of a complete component. This is an example of a situation that will be taken care of by the type system presented in the next section, to ensure that the coercion, in Rule (Eval Compose), from $\mathsf{conf}(c)$ to $\mathsf{comp}(c)$ is sound.

The evaluation of $\mathsf{new}\ e$ uses the composition operation stored in the component value yield by expression e as the building instructions for the new object. Rule (Eval New) expresses that the subexpression e evaluates to a component, and that its internal composition operations are applied to an empty object instance. This is expressed by the premise $\mathbf{0}; c; S \Downarrow s; S'$ where s is the resulting object. The instance construction produces an object with possibly unsatisfied required ports, which are then satisfied by plug-assignments. The connections are directly performed in the heap by using the locations corresponding to the required ports.

The second judgement $s; c; S \Downarrow s'; S'$, defines the application of composition operations to composition contexts as defined by the rules in Figure 3.5. This construction is achieved by manipulating the records r, e, and p that make up the splitted view of the object, $s = (r, e, p)$.

In order to bring explicit the role of composition operations in the semantic rules we find convenient to introduce an alternative representation of objects where, for convenience we group separately required ports, provided ports, and internal elements. We define the splitted view of an object as follows:

Definition 3.6 (Splitted View). *Given an object $s = \{\ell_i = l_i{}^{i \in 1..n}\}$, its splitted representation is a tuple of records (r, e, p) such that r, e, and p are disjoint and contain all the labels in s.*

This notion of splitted object is such that r represents the required ports, e the internal elements, and p the provided ports of the object. We use one representation or the other interchangeably whenever convenient and use $- \oplus -$ to denote the concatenation of the records. For the sake of simplicity we assume that the labels that are common to both records take the value of the second record in the concatenation.

As expected, Rule (App Sequence) sequentially applies the two parts of the operation thus causing the combined effect of both. For the ports, Rules (App Requires) and (App Provides) both create empty placeholders in the heap and make the correspondence with local names in the records r or p.

The introduction of an internal component corresponds, at the instance level, to the introduction of an instance of that component inside the object currently being built. So, the integration proceeds by instantiating the internal component and introducing the resulting object as an internal element of the instance, in record e, as specified by Rule (App Uses). Remember that the application of a configurator is only performed with ground composition operations (taken from a configurator value) and therefore their internal expression are values. In this case this value is expected to be a component. Similarly, (App Method Block) takes the field values, builds a record associated with the declared local name, and links it into the composition context. In Rule (App Method Block), $v_i[(r, e, p)]$ denotes the substitution of the object's labels by their locations and therefore give access to the elements already in the instance (see page 44). The substitution $[x \leftarrow \imath]$ introduces the "self" reference of the method block. Notice that the application of substitutions to method block expressions disallows the replacement of these names. Thus, they remain bound until applied to a particular composition context.

Finally, the application of a plug expression connects source to target ports by simply forming a chain between the two locations, Rule (App Plug). We use the function $\mathrm{select}_S(o, \pi)$ to denote the location corresponding to port π. In the case of an undefined location for π_2 the we consider that the heap is not changed, it corresponds to trying to connect an implementation to a place where it is not needed. If π_1 is undefined then a run-time error would occur, but this is trapped by our type system as we show in our subject reduction result.

The internal elements of an object may be method blocks and ports whose names are referred directly and there are ports of internal elements that are only accessible by dereferencing a local name. To abbreviate the writing of the rules that connect ports inside an object, we use an auxiliary operation $\mathrm{select}_{-}(-,-)$ to locate the placeholder (the location) of a port in an object, given its name (simple or compound). The operation $\mathrm{select}_{-}(-,-)$ is defined as follows:

Definition 3.7 (Port selection)**.** *Given any closed heap S, any object o, and a port name π*

$$\mathrm{select}_S(o, \pi) \triangleq \begin{cases} \imath & \textit{if } o = \{x = \imath, \ldots\} \textit{ and } \pi = x \\ \imath' & \textit{if } o = \{x = \imath, \ldots\},\ \pi = x.\ell, \textit{ and } S(\imath) = \{\ell = \imath', \ldots\} \\ \textit{undefined} & \textit{otherwise} \end{cases}$$

$\mathrm{select}_S(o, \pi)$ denotes the location of port π in the context of object o with relation to S. This concludes our presentation of the operational semantics of the untyped component calculus λ_χ.

Notice that objects are represented by linked structures constituted of ports, other objects, and records containing variables and methods.

3.1.1 Remarks

We now discuss some issues related to our language that are worth noting. Notice that the structures denoted by configurators, besides being the raw material and result of λ_χ computations, are mapped onto the structures of component instances. The elements of these webs of objects are interconnected by chains of memory locations, leading from a provided port of an instance, to an actual record containing methods. The intermediate redirections in the heap represent nested layers of internal elements. By using a "chain-transparent" selection operation, we are able to access methods via these chains of locations without explicitly referring them. In a optimised implementation, $\mathrm{deref}_{-}(-)$ can be suppressed by collapsing the chains at instantiation time thus allowing for a direct access to methods.

Notice that the provided ports of an object may lead directly to implementations outside the current object, if explicitly connected to one of its required ports. Although apparently not necessary from a small scale programming perspective, this pattern is, for sure, interesting from a component construction perspective, where elements are chosen according to run-time demands.

We require, in the syntax of method blocks, fields to be initialised using values which in our case must be abstractions, thus disallowing the usage of arbitrary unevaluated expressions. Method blocks are configurators denoting declarations whose initial values are yield by expressions with bound occurrences of variables. The intended scope of those variables is a composition context yet to be defined by composition with other configurators. Evidence of this scoping mechanism appears in the definition of the application of substitutions to expressions, Definition 3.3 in the case of method blocks $x_L[...]$ where the name of the method block (x) and the variables declared in set (L) are not replaced. It is also visible in Rule (App Method Block) when local names of the partially built objects get replaced in method bodies. The evaluation of these field initialiser expressions is delayed until the composition context gets well-formed, that will happen only after the instantiation of the component, when all names get bound to the concrete appropriate elements of that object. This mechanism can be seen as a kind of disciplined dynamic scoping which is safe with relation to the typing relation defined in the typed version of the component calculus. The syntactic restriction imposed by our design, to use only closed values as initialiser expressions, greatly simplifies the definition of the semantics in this regard.

Notice also that the evaluation relation for computational expressions ($e; S \downarrow v; S$) depends on the evaluation relation for application of configurators ($s; c; S \Downarrow s; S$), but not conversely (the dependency through new is not essential). In particular it is not possible to embed computations yielding values in the middle of computations assembling or linking objects. This is a necessary condition to avoid undefined computations such as calling on a port which is not, at that stage, connected to an appropriate resource. This closely relates to the phase distinction studied in [22], as we need to separate the evaluation of configurator values (run-time phase), from the instantiation of components (compile-link-time phase), from the execution of methods (run-time phase again).

3.1.2 An Example

In order to give some preliminary evidence of the expressiveness of our language, we now show how it can emulate standard object-oriented language mechanisms such as implementation inheritance and mixins. We use standard notation for **let** constructions and λ abstractions (**fun** x $\rightarrow$ ...).

Example 3.8. In this example, we illustrate the encoding of classes and mixins using components with explicit dependencies. We use a required port on a component to represent the reference to self in the definition of a mixin-like component. Consider component c implementing methods m1 and m2 which are available at a provided port self, and based on a required port super,

```
let c = compose (requires super;
                 provides self;
                 m[m1 = fun x→x+1, m2 = fun x→2*super.m1(x)];
                 plug m into self) in...
```

Notice that m2 depends on another method, m1, but that this is called on an "open" self reference, available at the required port super. Component c can be seen as a "mixin" which we can "close" by means of an extra composition layer to produce a "class" component C:

```
let C = compose (provides self;
                 x[c];
                 plug x.self into x.super;
                 plug x.self into self) in ...
```

We instantiate component C,

```
let o = new C in o.self.m2(1)
```

to obtain an object where method m2 refers to method m1 in the same object. We consider that component c is a "mixin" and that its closed version, component C, is a "class". Now, to extend c in order to log the calls to its methods, we define a "mixin" component log (for the sake of simplicity consider the existence of a primitive function print):

```
let log = compose (requires super;
                   provides self;
                   m[m1 = fun x →(print("m1"); super.m1(x)),
                     m2 = fun x →(print("m2"); super.m2(x))];
                   plug m into self) in...
```

and apply the extension log to c in the following composition

```
let logc = compose(requires super;
                   provides self;
                   log[log]; c[c];
                   plug super into c.super;
                   plug c.self into log.super;
                   plug log.self into self) in...
```

which again creates a "mixin", and by a simple composition operation we obtain a "class"

```
let Log = compose (provides self;
                   x[logc];
```

```
                    plug x.self into x.super;
                    plug x.self into self) in ...
```

which gets instantiated as follows:

```
  let o = new Log in o.self.m2(1)
```

The calls made at port self are directed to log and then to c, e.g. the call o.self.m2(1) is directed to log, which prints the string "m2" in the console, and follows to m2 in c, which in turn calls m1 in log, thus printing "m1" and, from c, returns 2, and from m2 returns 4.

Although requiring some syntactic sugar, to close all mixin components into classes, this example shows that our component language is expressive enough to encode classes and mixins.

To complete our first language proposal, we next describe a type system that ensures the consistency of all these operations at compile-time.

3.2 λ_χ^τ — A Typed Component Calculus

The language λ_χ defined in the previous section supports a clear definition of the architecture of a system, which would otherwise be encoded in non-specific construction code. Furthermore, by treating architectural building blocks, components and configurators, as first-class values, our language allows us to express the construction of systems in a dynamic and computation-driven way (instead of the traditional static linking mechanism of module languages).

In a typeful programming setting, the usage of composition abstractions should also allow for the automatic verification of architectural properties. Our approach is to assign static type information to configurators, components and objects and type each composition operation accordingly. Besides the extensional type information of components and objects, which specify their interfaces, we use intensional type information about the internal structure of configurators. We refer this information as intensional because it talks about the internal structure of components rather than about their functionality.

Our type system ensures, not only the type safety for computations, as expected from an extension of the typed version of the base language (λ_R), but also for compositions. Intuitively, we say that a composition is well-typed only if all the declared services are indeed implemented and if all dependencies of its internal elements are properly satisfied. In particular, no "null reference errors" may occur due to the absence of properly linked resources. Once this basic property is recursively ensured throughout the hierarchy of components we know that no execution error will occur because of a badly assembled component.

In this section we define, by extension of the type system of λ_R^τ, a type system to ensure this architectural soundness. To that end, we first define a type language that captures the se-

$$
\begin{array}{lcll}
\tau & ::= & & \text{types} \\
 & | & \tau \to \tau & \text{function type} \\
 & | & \{|\ell_i : \tau_i \ ^{i \in 1..n}|\} & \text{record type} \\
 & | & \{\ell_i : \tau_i \ ^{i \in 1..n}\} & \text{interface type} \\
 & | & \tau \Rightarrow \tau & \text{component type} \\
 & | & \{r_i \ ^{i \in 1..n}\} \Longrightarrow \{r_i \ ^{i \in 1..m}\} & \text{configurator type} \\
 & & & \\
r & ::= & & \text{resources} \\
 & | & \pi \circ \tau & \text{unsatisfied resource} \\
 & | & \pi \bullet \tau & \text{available resource} \\
 & | & \pi \triangleright \tau & \text{provided resource} \\
 & | & \pi \triangleleft \tau & \text{required resource}
\end{array}
$$

Figure 3.6: Abstract syntax of λ_χ^τ types.

mantics of the new language constructs, and then extend λ_χ with type annotations, and define a typing relation for the language expressions. Type safety is finally proven in a subject reduction theorem that correlates the semantics of the language, using both evaluation judgements, and the typing information statically assigned to the expressions.

Types

We first define a type language by extending the types of λ_χ, $\mathcal{T}_R$ (Definition 2.13), with new types for configurators, components, objects, and ports.

Definition 3.9 (Types)**.** *The types $\mathcal{T}_\chi$ of λ_χ^τ are defined by the abstract syntax in Figure 3.6.*

Besides the types of the base language λ_R^τ, the type language $\mathcal{T}_\chi$ includes new type forms for ports, objects, components and configurators. While record types in $\mathcal{T}_R$ follow the usual definition for typing mutable records, a distinctive annotation, interface types, is used here to type records when encapsulation of objects and ports is intended: interfaces provide a read-only perspective on record values. Port and object values are both encoded with records, and they should be protected from external modification, hence we type them using interface types.

Port types follow the standard definition of interface types in object-oriented programming languages as collections of methods: names are associated with function types.

Object types, which are collections of named port types, are also encoded by interface types. They describe the public interface of component instances, as well as the provided and required ports of component values.

Component types, of the form $\tau \Rightarrow \sigma$, characterise component values with the object type τ specifying the names and types of the required ports, whereas the object type σ specifies the provided ports. The target type σ is the object type assigned to the instances that such components produce.

Configurator types carry information about the effects of configurators on compositions. This is expressed in the form of required and provided *resources* (do not confuse with required and provided service *ports*). A resource is represented by a combination of a tag, a name, and a type. The possible tags are: $\circ$ (open), meaning that the resource is unsatisfied, for instance, that a provided port is not connected; $\bullet$, meaning that the resource is available for connection, for instance a certain method block or internal component is present; $\triangleright$ denotes that a provided port is present, and $\triangleleft$ denotes that a required port is present. Typically, at the level of typing, a composition operation rewrites a bag of resources into another bag of resources, reflecting the internal change that takes place in the component architecture.

Notation 3.10. In general, we use the symbol K to denote resource sets. We also define K_* to be the resource set containing all resources tagged with the mark $*$ in K where $*$ may be $\circ$, $\bullet$, $\triangleright$, or $\triangleleft$. For example, $K_\bullet$ is $\{\pi_i : \tau_i\ ^{i\in 1..n}\}$ where $\pi_i \bullet \tau_i$ for $i = 1..n$ are all the $\bullet$-tagged elements in K. In appropriate situations where, in all resources $\pi_i * \tau_i$, π_i are labels ℓ_i, we use K_* as an interface type assigning types τ_i to labels ℓ_i. We use I, J for interface types and R, P for object types, i.e. interfaces of the form $\{\ell_i : I_i\ ^{i\in 1..n}\}$. We denote by $- \oplus -$ the concatenation operation on disjoint interfaces, and by $-\#-$ the disjointness predicate for interfaces and resource sets. When convenient, we abbreviate $\ell \bullet \tau, K$ for the set of resources $\{\ell \bullet \tau\} \cup K$.

These notions are formally defined as follows:

Definition 3.11 (Operations on interfaces and resource sets)**.**

- *If* $K = \{\pi_i \bullet \tau_i\ ^{i\in 1..n_1}, \pi'_i \circ \tau'_i\ ^{i\in 1..n_2}, \pi''_i \triangleright \tau''_i\ ^{i\in 1..n_3}, \pi'''_i \triangleleft \tau'''_i\ ^{i\in 1..n_4}\}$ *then*
  - $K_\bullet \triangleq \{\pi_i : \tau_i\ ^{i\in 1..n_1}\}$
  - $K_\circ \triangleq \{\pi'_i : \tau'_i\ ^{i\in 1..n_2}\}$
  - $K_\triangleright \triangleq \{\pi''_i : \tau''_i\ ^{i\in 1..n_3}\}$
  - $K_\triangleleft \triangleq \{\pi'''_i : \tau'''_i\ ^{i\in 1..n_4}\}$
- $\{\ell_i : \tau_i\ ^{i\in 1..n}\} \oplus \{\ell'_j : \tau'_j\ ^{j\in 1..m}\} \triangleq \{\ell_i : \tau_i\ ^{i\in 1..n}, \ell'_j : \tau'_j\ ^{j\in 1..m}\}$
- $\{\ell_i : \tau_i\ ^{i\in 1..n}\}\ \#\ \{\ell'_j : \tau'_j\ ^{j\in 1..m}\}$ *if* $\ell_i \neq \ell'_j\ \ \forall_{i\in 1..n}\ \forall_{j\in 1..m}$
- $\{\ell_i * \tau_i\ ^{i\in 1..n}\}\ \#\ \{\ell'_j ** \tau'_j\ ^{j\in 1..m}\}$ *if* $\ell_i \neq \ell'_j\ \ \forall_{i\in 1..n}\ \forall_{j\in 1..m}$ *and* $*, ** \in \{\bullet, \circ, \triangleleft, \triangleright\}$
- $\{\ell_i : \tau_i\ ^{i\in 1..n}\} \subseteq \{\ell_i : \tau_i\ ^{i\in 1..n}, \ell'_j : \tau'_j\ ^{j\in 1..m}\}$

Notice that, in $\mathcal{T}_\chi$, not all expressions denote meaningful types, for instance, in a component type of the form $\tau \Rightarrow \sigma$, τ and σ are expected to be object types, expressing the required

e	$::=$		terms
	$\vert$	x	variable
	$\vert$	$\lambda x : \tau . e$	abstraction
	$\vert$	$e(e)$	application
	$\vert$	$\{\ell_i = e_i \; {}^{i \in 1..n}\}$	record
	$\vert$	$e.\ell$	selection
	$\vert$	$e.\ell := e$	assignment
	$\vert$	ℓ	port label
	$\vert$	$\mathsf{compose}\ e$	component creation
	$\vert$	$\mathsf{new}\ e\ \mathsf{with}\ \ell_j := e_j \; {}^{j \in 1..m}$	instantiation
	$\vert$	$\mathsf{requires}\ \ell : \tau$	required port
	$\vert$	$\mathsf{provides}\ \ell : \tau$	provided port
	$\vert$	$x[e : \tau]$	component introduction
	$\vert$	$x_I[\ell_i : \tau_i = \lambda x : \tau . e_i \; {}^{i \in 1..n}]$	method block
	$\vert$	$\mathsf{plug}\ \pi : \tau\ \mathsf{into}\ \pi : \tau$	plug
	$\vert$	$e ; e$	configurator composition
	$\vert$	ι	location value
	$\vert$	nil	null value
	$\vert$	$\mathsf{conf}(e)$	configurator
	$\vert$	$\mathsf{comp}(e)$	component
π	$::=$	$\ell \mid x \mid x.\ell$	port name

Figure 3.7: Abstract syntax of λ_χ^τ.

and provided services of the component: the type system will only accept meaningful type expressions.

Given this type language we now define the typed component calculus.

Syntax

We extend our untyped component calculus (λ_χ) with type annotations as follows:

Definition 3.12 (Terms)**.** *The language λ_χ^τ is defined by the abstract syntax in Figure 3.7.*

Port declarations, in the expressions $\mathsf{requires}\ \ell : \tau$ and $\mathsf{provides}\ \ell : \tau$, are decorated with type annotations to specify the type of the corresponding port — remember that the type of a component is built from its port names and types.

The introduction of an internal component, $x[e : \tau]$, also statically declares its type.

The declaration of a method block, $x_I[\ell_i : \tau_i = \lambda x : \tau . e_i \; {}^{i \in 1..n}]$, mentions, besides the names, the types of the expected elements in the composition context where it is introduced (I). It

$$
\begin{array}{lcll}
v & ::= & & \text{values} \\
 & | & \lambda x:\tau.e & \text{abstraction} \\
 & | & \{\ell_i = l_i{}^{i\in 1..n}\} & \text{record} \\
 & | & l & \text{location value} \\
 & | & \mathsf{nil} & \text{null value} \\
 & | & \mathsf{conf}(c) & \text{configurator} \\
 & | & \mathsf{comp}(c) & \text{component} \\
 & & & \\
c & ::= & & \text{ground composition operations} \\
 & | & \mathsf{requires}\ \ell:\tau & \text{required port} \\
 & | & \mathsf{provides}\ \ell:\tau & \text{provided port} \\
 & | & x[v:\tau] & \text{component introduction} \\
 & | & x_I[\ell_i:\tau_i = \lambda x_i:\tau_i.e_i{}^{i\in 1..n}] & \text{method block} \\
 & | & \mathsf{plug}\ \pi:\tau\ \mathsf{into}\ \pi:\tau & \text{plug} \\
 & | & c;c & \text{composition} \\
 & & & \\
\pi & ::= & \ell \mid x \mid x.\ell & \text{port name}
\end{array}
$$

Figure 3.8: Abstract syntax of λ^{τ}_{χ} values.

also explicitly types its fields (τ_i). This explicit typing of fields makes it simpler typing method blocks as recursive definitions (the local name x may occur in each e_i). The inference of such types is not a difficult task, but for the sake of simplicity we choose to add explicit type annotations. Whenever possible, to avoid cluttering the presentation with heavy notation we omit the index I on method block when the imported names are clear from context. The ports in plug expressions are also annotated with the expected types for the connecting ports.

Given the language syntax, the evaluation results of λ^{τ}_{χ} are defined as expected, again defining configurator and component values on ground composition operations as in Definition 3.1.

Definition 3.13 (Values)**.** *The set $\mathcal{U}^{\tau}_{\chi} \subseteq \lambda^{\tau}_{\chi}$ is defined by the abstract syntax in Figure 3.8.*

We now formally define the operational semantics of λ^{τ}_{χ}.

Operational semantics

As in λ^{τ}_{R}, for the sake of completeness we now present the rule system that defines the operational semantics of λ^{τ}_{χ} in Figures 3.9, and 3.10 and refer to the rules in Figure 2.8 for the remaining constructs. The operational semantics of λ^{τ}_{χ} is defined as follows:

Definition 3.14 (Evaluation)**.** *Let $e \in \lambda^{\tau}_{\chi}$ and a heap S such that $(e;S)$ is a valid configuration. The evaluation relation of an expression e to a value v, written $e;S \downarrow v;S'$ is defined inductively by the rules in Figures 2.8, 3.9, and 3.10.*

$$\text{(Eval Compose)}\quad \frac{e; S \downarrow \mathsf{conf}(c); S'}{\mathsf{compose}\ e; S \downarrow \mathsf{comp}(c); S'}$$

$$\text{(Eval New)}\quad (s = (r, e, p),\quad r = \{\ell_j \mapsto \iota_j\ {}^{j \in 1..m}\},\ \iota = \text{new}(S))$$

$$\frac{e; S \downarrow \mathsf{comp}(c); S_0 \quad \mathbf{0}; c; S_n \Downarrow s; S_{n+1} \quad e_i; S_{i-1} \downarrow v_i; S_i\ \forall_{i \in 1..n}}{\mathsf{new}\ e\ \mathsf{with}\ \ell_i := e_i\ {}^{i \in 1..n}; S \downarrow \iota; S_{n+1}[\iota \mapsto s][\iota_i \mapsto v_i\ {}^{i \in 1..n}]}$$

$$\text{(Eval Requires)}^{\tau}\quad \mathsf{requires}\ \ell{:}\tau; S \downarrow \mathsf{conf}(\mathsf{requires}\ \ell{:}\tau); S$$

$$\text{(Eval Provides)}^{\tau}\quad \mathsf{provides}\ \ell{:}\tau; S \downarrow \mathsf{conf}(\mathsf{provides}\ \ell{:}\tau); S$$

$$\text{(Eval Plug)}^{\tau}\quad \mathsf{plug}\ \pi_1{:}\tau_1\ \mathsf{into}\ \pi_2{:}\tau_2; S \downarrow \mathsf{conf}(\mathsf{plug}\ \pi_1{:}\tau_1\ \mathsf{into}\ \pi_2{:}\tau_2); S$$

$$\text{(Eval Sequence)}\quad \frac{e_1; S \downarrow \mathsf{conf}(c_1); S' \quad e_2; S \downarrow \mathsf{conf}(c_2); S'}{(e_1; e_2); S \downarrow \mathsf{conf}(c_1; c_2); S'}$$

$$\text{(Eval Uses)}^{\tau}\quad \frac{e; S \downarrow v; S'}{x[e{:}\tau]; S \downarrow \mathsf{conf}(x[v{:}\tau]); S'}$$

$$\text{(Eval Method Block)}^{\tau}\quad x_I[\ell_i{:}\tau_i = v_i\ {}^{i \in 1..n}]; S \downarrow \mathsf{conf}(x_I[\ell_i{:}\tau_i = v_i\ {}^{i \in 1..n}]); S$$

Figure 3.9: Evaluation rules for λ^{τ}_{χ}.

Notice that the operational semantics of λ^{τ}_{χ} is the same as the one of λ_{χ}, just a minor change in the abstract syntax was introduced. Type annotations were introduced in order to guide the type system presented in the next section and do not play any role in the evaluation relation. Thus, the results that assert that the evaluation relation is well-defined (Lemma 3.5) still apply to the current context without modifications.

Type System

We now present a type system for λ^{τ}_{χ}. This is done by extending the type system of λ^{τ}_{R} with rules for typing the computational expressions $\mathsf{compose}\ e$ and $\mathsf{new}\ e$, rules for typing the composition expressions $\mathsf{requires}\ e$, $\mathsf{provides}\ e$, $x[-]$, $x_{-}[- = -]$, $\mathsf{plug}\ \pi\ \mathsf{into}\ \pi$, and $-;-$, as well as rules for typing the configurator and component values. We also adapt judgements $\Delta \vdash \diamond$ and $\Delta \vdash \tau\ \mathsf{ok}$ to address the new type forms for interfaces, configurator and component types. The appropriate definition for a valid typing environment is therefore defined by:

Definition 3.15 (Valid Typing Environment)**.** *A typing environment Δ is* valid *if the judgement $\Delta \vdash \diamond$ is derivable by the rules in Figures 2.10 and 3.11.*

Notation 3.16. In Rule (Type Conf) we write $(\ell : \tau) \in K$ to denote $\ell * \tau \in K$ where $*$ is one of the possible resource tags: $\circ$, $\bullet$, $\triangleleft$, or $\triangleright$.

Our type system enforces encapsulation (information hiding) of objects and components in the sense already discussed. Namely, port names and types in a component border are the only information exported to the outside, thus defining a visibility boundary ensuring that

$$\text{(App Requires)}^{\tau} \qquad\qquad (\imath = \text{new}(S))$$
$$(r,e,p); (\textsf{requires}\ \ell:\tau); S \Downarrow (r \oplus \{\ell = \imath\}, e, p); S[\imath \mapsto \textsf{nil}]$$

$$\text{(App Provides)}^{\tau} \qquad\qquad (\imath = \text{new}(S))$$
$$(r,e,p); (\textsf{provides}\ \ell:\tau); S \Downarrow (r, e, p \oplus \{\ell = \imath\}); S[\imath \mapsto \textsf{nil}]$$

$$\text{(App Uses)}^{\tau}\ \frac{\textsf{new}\ v; S \downarrow \imath; S'}{(r,e,p); x[v:\tau]; S \Downarrow (r, e \oplus \{x = \imath\}, p); S'} \qquad \text{(App Sequence)}\ \frac{s; c_1; S \Downarrow s'; S' \quad s'; c_2; S' \Downarrow s''; S''}{s; (c_1; c_2); S \Downarrow s''; S''}$$

$$\text{(App Method Block)}^{\tau} \qquad (\imath,\ \imath_i\ {}^{i \in 1..n} = \text{new}(S), \quad v'_i = v_i[(r,e,p)][x \leftarrow \imath])$$
$$(r,e,p); x_I[\ell_i : \tau_i = v_i\ {}^{i \in 1..n}]; S \Downarrow (r, e \oplus \{x = \imath\}, p); S[\imath \mapsto \{\ell_i = \imath_i\ {}^{i \in 1..n}\}][\imath_i \mapsto v'_i\ {}^{i \in 1..n}]$$

$$\text{(App Plug)}^{\tau}$$
$$s; \textsf{plug}\ \pi_1 : \tau_1\ \textsf{into}\ \pi_2 : \tau_2; S \Downarrow s; S[\text{select}_S(s, \pi_2) \mapsto \text{select}_S(s, \pi_1)]$$

Figure 3.10: Application rules for composition operations in λ^{τ}_{χ}.

$$\text{(Type Interface)}\ \frac{\Delta \vdash \tau_i\ \textsf{ok} \quad \forall_{i \in 1..n}}{\Delta \vdash \{\ell_i : \tau_i\ {}^{i \in 1..n}\}\ \textsf{ok}} \qquad \text{(Type Comp)}\ \frac{\Delta \vdash \tau\ \textsf{ok} \quad \Delta \vdash \sigma\ \textsf{ok}}{\Delta \vdash \tau \Rightarrow \sigma\ \textsf{ok}} \qquad \text{(Type Conf)}\ \frac{\Delta \vdash \tau\ \textsf{ok} \quad \forall_{\tau}.(\ell : \tau) \in K \cup K'}{\Delta \vdash K \Rightarrow K'\ \textsf{ok}}$$

Figure 3.11: Validation rules for typing environments in λ^{τ}_{χ}.

components and configurators resulting from evaluation are pure values. To that end, we restrict the typing environment in the typing of subexpressions of the method blocks, Rule (Comp Method Block), by explicitly disallowing any reference to imperative values. External references are only allowed for declarative stateless values, i.e. only configurators and components can be referred in the definition of other configurators or components. Such restriction on typing environments is defined as follows:

Definition 3.17 (Restricted Typing Environment). *Let Δ be a typing environment, $|\Delta|$ is the typing environment containing the elements of Δ with component or configurator types:*

$$|\Delta| \triangleq \{x : \tau \mid x : \tau \in \Delta \textit{ and } \tau = (\sigma \Rightarrow \sigma') \textit{ or } \tau = (K \Longrightarrow K')\}$$

We can now define the typing relation as follows:

Definition 3.18 (Typing relation). *The judgement $\Delta \vdash e : \tau$ is valid if it is derivable by the rules in Figures 2.11, 3.12 and 3.13.*

In our type system, we extend the typing relation of λ^{τ}_{R}, defined by the rules in Figure 2.11 with typing rules for the new constructs. We also define an additional typing for record values using

$$\text{(Val Interface)}\quad (m \leq n) \qquad\qquad \text{(Val Select Interface)}$$

$$\frac{\Delta \vdash e : \{\!|\ell_i : \tau_i \;^{i \in 1..n}|\!\}}{\Delta \vdash e : \{\ell_i : \tau_i \;^{i \in 1..m}\}} \qquad\qquad \frac{\Delta \vdash e : \{\ldots, \ell : \tau, \ldots\}}{\Delta \vdash e.\ell : \tau}$$

$$\text{(Val Compose)}\quad (K_{\circ} = \emptyset) \qquad\qquad \text{(Val New)}$$

$$\frac{\Delta \vdash e : \emptyset \Longrightarrow K}{\Delta \vdash \mathsf{compose}\ e : K_{\triangleleft} \Rightarrow K_{\triangleright}} \qquad\qquad \frac{\Delta \vdash e : \{\ell_i : \tau_i \;^{i \in 1..n}\} \Rightarrow \sigma \quad \Delta \vdash e_i : \tau_i \quad \forall_{i \in 1..n}}{\Delta \vdash \mathsf{new}\ e\ \mathsf{with}\ \ell_i := e_i \;^{i \in 1..n} : \sigma}$$

$$\text{(Val Composition Value)} \qquad\qquad \text{(Val Configurator Value)}$$

$$\frac{\Delta \vdash \mathsf{compose}\ c : \tau \Rightarrow \sigma}{\Delta \vdash \mathsf{comp}(c) : \tau \Rightarrow \sigma} \qquad\qquad \frac{\Delta \vdash c : K \Longrightarrow K'}{\Delta \vdash \mathsf{conf}(c) : K \Longrightarrow K'}$$

Figure 3.12: Typing rules for λ^{τ}_{χ}.

interface types. Namely, we introduce Rule (Val Interface) that realises a type coercion from a record type to an interface type. This is a restricted form of subsumption relating records and interfaces.

We then define a selection rule based on interfaces, Rule (Val Select Interface). The combination of these two rules enforces the read-only perspective that interface types provide on record values. For instance, method blocks evaluate to record values, are typed as records inside their methods, but are typed as interfaces when used by other elements or exported by objects. We disallow assignments on values typed as ports, and in this way encapsulate state. The side condition of this rule ($m \leq n$) also allows some fields of a record to be hidden.

The expressions, $\mathsf{compose}\ e$ and $\mathsf{new}\ e$, are typed together with composition expressions, to ensure the soundness of components, configurators and objects. We next describe in detail the typing rules of the language, which, besides manipulating the extensional type information of components and objects, also relies on configurator types that carry intensional type information, describing the effects that configurators produce. This information is then combined in the typing of configurator composition. The overall result is that well-typed configurators always produce well-formed structures of objects in the sense that all dependencies and specified provided services are implemented, that is the architectures resulting from the evaluation process are sound as discussed in Section 1.3.4. In particular, this soundness requires the absence of broken chains of locations in the heap.

The type system defines three levels of information hiding and three levels of architectural soundness for the three different sorts of values.

(Comp Requires)

$$\Delta \vdash (\mathsf{requires}\ \ell : \tau) : \emptyset \Longrightarrow \{\ell \bullet \tau, \ell \triangleleft \tau\}$$

(Comp Provides)

$$\Delta \vdash (\mathsf{provides}\ \ell : \tau) : \emptyset \Longrightarrow \{\ell \circ \tau, \ell \triangleright \tau\}$$

(Comp Plug)

$$\Delta \vdash \mathsf{plug}\ (\pi_1 : \tau)\ \mathsf{into}\ (\pi_2 : \tau) : (\{\pi_2 \circ \tau, \pi_1 \bullet \tau\} \Longrightarrow \{\pi_1 \bullet \tau\})$$

(Comp Sequence) $(K' \# K'', K' \# K''')$

$$\frac{\Delta \vdash e_1 : K \Longrightarrow K', K_c \quad \Delta \vdash e_2 : K_c, K'' \Longrightarrow K'''}{\Delta \vdash (e_1; e_2) : K, K'' \Longrightarrow K', K'''}$$

(Comp Uses) $(\tau = \{\ell_i^r : \tau_i{}^{i \in 1..n}\},\ \sigma = \{\ell_j^p : \sigma_j{}^{j \in 1..m}\})$

$$\frac{\Delta \vdash e : \tau \Rightarrow \sigma}{\Delta \vdash x[e : \tau \Rightarrow \sigma] : \emptyset \Longrightarrow \left\{x \bullet \sigma, x.\ell_i^r \circ \tau_i{}^{i \in 1..n}, x.\ell_j^p \bullet \sigma_j{}^{j \in 1..m}\right\}}$$

(Comp Method Block) $(I = \{\ell_i' : \tau_i'{}^{i \in 1..m}\},\ K = \{\ell_i' \bullet \tau_i'{}^{i \in 1..n}\})$

$$\frac{|\Delta|, \ell_i' : \tau_i'{}^{i \in 1..m}, x : \{\!|\ell_i : \tau_i{}^{i \in 1..n}|\!\} \vdash e_i : \tau_i \quad \forall_{i \in 1..n}}{\Delta \vdash x_I[\ell_i : \tau_i = e_i{}^{i \in 1..n}] : K \Longrightarrow K, \{x \bullet \{\ell_i : \tau_i{}^{i \in 1..n}\}\}}$$

Figure 3.13: Typing rules for λ_χ^τ.

Configurators are typed so that their composition is statically verifiable, i.e. their structure is made sufficiently visible in the given type, and the type of the resulting configurator is obtained by composition of the two given types.

Components are typed to allow the static verification of black-box compositions. The usage of components on an introduction operation of the form $x[-]$ inside other compositions only makes available for further use their required and provided ports.

Finally, objects are typed in the perspective of direct utilisation, where only the provided ports are disclosed so that ports and methods are accessed properly. These points are visible in the typing rules of Figures 3.12 and 3.13, which we now explain in more detail.

Rule (Val Compose) ensures that components are only produced given completed architectures, i.e. from configurators that do not depend on any existing resource ($\emptyset \Longrightarrow K$) and does not introduce unsatisfied resources ($K_\circ = \emptyset$). These conditions, in combination with the typing of component and configurator values are important to ensure the soundness of architectures, thus preventing run-time errors from occurring. The resulting component type, $K_\triangleleft \Rightarrow K_\triangleright$, reveals only the required and provided services, while hiding the remaining (intensional) information about the internal structure of the component.

Rule (Val New) types a new instance with the object type containing the provided ports of its generator component and checks for the proper satisfaction of all required ports, if there are any. Notice that, once again, some type information gets erased: here, the existing required ports are not included in the instance type. Hence, the type system does not distinguish instances that provide the same services, as expected. We now turn to composition operations.

Our type system assigns a type of the form $K \Longrightarrow K'$ to each composition operation, in order to denote the transformation of a set of required resources (K) into a set of provided resources (K'). These typings of configurators are the base for the operations that generate components and objects, typed by Rules (Val Compose) and (Val New). Basic composition operations get natural configurator types, which are then elaborated by means of composition.

In Rule (Comp Provides), the type of (provides $\ell : \tau$) indicates that an unsatisfied resource of the form ($\ell \circ \tau$) is provided, i.e. a resource that must be satisfied before this configurator is used to make a component, and of a new provided port ($\ell \triangleright \tau$).

The symmetrical Rule (Comp Requires), (requires $\ell : \tau$) adds a new required port ($\ell \triangleleft \tau$) and an available resource ($\ell \bullet \tau$) to a composition context.

The typing of the introduction operation of an internal component ($x[e : \tau]$), described in Rule (Comp Uses), indicates that it provides available resources corresponding to an instance of the internal component ($x \bullet \{\ell_j^p : \sigma_j{}^{j \in 1..m}\}$) and its provided ports ($x.\ell_j^p \bullet \sigma_j{}^{j \in 1..m}$), and unsatisfied resources that denote the internally required ports ($x.\ell_i^r \circ \tau_i{}^{i \in 1..n}$).

Similar type information is associated with method blocks introduced by ($x_I[\ell_i : \tau_i = e_i{}^{i \in 1..n}]$), in Rule (Comp Method Block), but using the required set of resources obtained from I as a precondition to its application. In this case, the method block itself is provided as an available resource ($x \bullet \{\ell_i : \tau_i{}^{i \in 1..n}\}$).

Rule (Comp Sequence) combines the effect of two expressions. This rule depicts the propagation of resources (K_c) from e_1 to e_2, meaning that e_2 handles these resources either by keeping them in K''' or by consuming them.

Notice that, on the typing of method blocks, each field is typed in a context where the corresponding method block is typed as record — and used as the self reference — but the method block itself is introduced in the composition context typed by an interface. Thus, fields are only modifiable from within the method block itself, and are read-only towards the rest of the elements in the component and consequently to the outer context.

As observed before, for a sequence of composition operations to be accepted in a compose expression it must denote a complete architecture, in particular the set of unsatisfied resources must be empty ($K_\circ = \emptyset$). The elimination of these resources from the configurator types is captured by the typing of plug operations: they are typed with a required resource ($\pi_2 \circ \tau$) that is not propagated to the set of provided resources, as made explicit in Rule (Comp Plug). This denotes the satisfaction of internal dependencies in a composition. When used in sequence with other operations, the unsatisfied resources are either eliminated, i.e. the dependency gets satisfied, or they are propagated to the required resources of the composition. The one-time-usage of an unsatisfied resource is the operation behind the intuition of the name "resource" given to the basic elements of configurator types.

Given this description of the type system, we next illustrate its use in typing a composition expression.

Example 3.19. Given the type abbreviations $\top = \{\}$, $\sigma = \top \rightarrow \top$, and $\tau = \{m : \sigma\}$, consider the following λ^{τ}_{χ} expression that defines a component that redirects all method calls from a required to a provided port, and instantiates it:

$$\begin{array}{l}\mathsf{let}\ c = \mathsf{compose}\ (\mathsf{provides}\ p:\tau;\ \mathsf{requires}\ q:\tau;\ \mathsf{plug}\ q:\tau\ \mathsf{into}\ p:\tau)\ \mathsf{in}\\ \mathsf{let}\ o = \mathsf{new}\ c\ \mathsf{with}\ q{:=}\{m = \lambda x:\top.x\}\ \mathsf{in}\ o.p.m(\{\})\end{array}$$

This means that all calls on port p are in fact dealt by the record plugged to port q. The typing for the composition expression $((\mathsf{provides}\ p:\tau; \mathsf{requires}\ q:\tau); \mathsf{plug}\ q:\tau\ \mathsf{into}\ p:\tau)$ is obtained by the following derivation:

(Comp Provides)	$\vdash (\mathsf{provides}\ p:\tau){:}\emptyset \Longrightarrow \{p \circ \tau, p \triangleright \tau\}$
(Comp Requires)	$\vdash (\mathsf{requires}\ q:\tau){:}\emptyset \Longrightarrow \{q \bullet \tau, q \triangleleft \tau\}$
(Comp Sequence)	$\vdash (\mathsf{provides}\ p:\tau; \mathsf{requires}\ q:\tau) : \emptyset \Longrightarrow \{q \triangleleft \tau, p \triangleright \tau, p \circ \tau, q \bullet \tau\}$
(Comp Plug)	$\vdash (\mathsf{plug}\ q:\tau\ \mathsf{into}\ p:\tau){:}\{p \circ \tau, q \bullet \tau\} \Longrightarrow \{q \bullet \tau\}$
(Comp Sequence)	$\vdash ((\mathsf{provides}\ p:\tau; \mathsf{requires}\ q:\tau); \mathsf{plug}\ q:\tau\ \mathsf{into}\ p:\tau) : \emptyset \Longrightarrow \{q \triangleleft \tau, p \triangleright \tau, q \bullet \tau\}$

Notice that the unsatisfied resource $p \circ \tau$ is eliminated from the provided resources by the plug expression. The second application of Rule (Comp Sequence) has $K_c = \{p \circ \tau, q \bullet \tau\}$ as the common set of resources whereas the provided resource set is only $\{q \bullet \tau\}$. So, the result of the type combination is given by the provided resources $\{q \triangleleft \tau, p \triangleright \tau, q \bullet \tau\}$.

Using this configurator in the compose expression is valid since the side conditions of Rule (Val Compose) are satisfied by $\emptyset \Longrightarrow \{q \triangleleft \tau, p \triangleright \tau, q \bullet \tau\}$. It has no required resources and adds no unsatisfied resources. This produces a component c with type $\{q:\tau\} \Rightarrow \{p:\tau\}$. So, we have the following typing derivation for the remainder of the expression:

(Val Var)	$c:\{q:\tau\} \Rightarrow \{p:\tau\} \vdash c:\{q:\tau\} \Rightarrow \{p:\tau\}$
(Val Var)	$c:\{q:\tau\} \Rightarrow \{p:\tau\}, x:\top \vdash x:\top$
(Val Abstraction)	$c:\{q:\tau\} \Rightarrow \{p:\tau\} \vdash \lambda x:\top.x:\sigma$
(Val Record)	$c:\{q:\tau\} \Rightarrow \{p:\tau\} \vdash \{m = \lambda x:\top.x\}:\tau$
(Val New)	$c:\{q:\tau\} \Rightarrow \{p:\tau\} \vdash \mathsf{new}\ c\ \mathsf{with}\ q{:=}\{m = \lambda x:\top.x\}:\{p:\tau\}$

which assigns type $\{p:\tau\}$ to o. Notice that the typing for new derives the object type from the type of the generating component, and that the requirements are verified. We now conclude that:

(Val Var)	$c:\{q:\tau\} \Rightarrow \{p:\tau\},\ o:\{p:\tau\} \vdash o:\{p:\tau\}$
(Val Select)	$c:\{q:\tau\} \Rightarrow \{p:\tau\},\ o:\{p:\tau\} \vdash o.p:\tau$
(Val Select)	$c:\{q:\tau\} \Rightarrow \{p:\tau\},\ o:\{p:\tau\} \vdash o.p.m:\top \rightarrow \top$
(Val Record)	$c:\{q:\tau\} \Rightarrow \{p:\tau\},\ o:\{p:\tau\} \vdash \{\}:\top$
(Val Application)	$c:\{q:\tau\} \Rightarrow \{p:\tau\},\ o:\{p:\tau\} \vdash o.p.m(\{\}):\top$

Notice that we have omitted the typing of the let expressions which are not part of our language but are easily encoded in the standard way.

This example illustrates our type system while validating compositions. Notice that the typing configurator composition is accomplished by the composition of the types of the parts. The result is then used to build the types of components and objects. We now combine the type system with the semantics of the language to prove type safety for λ_χ^τ.

3.2.1 Type Safety

In this section, we state, characterise, and prove type safety in λ_χ^τ. Type safety is a corollary of a subject reduction theorem that ensures the architectural soundness of configurators, components and instances as a side effect of the traditional progress and type preservation properties. By analogy with type safety result in λ_R^τ, we extend of the operational semantics with a distinguished value wrong, to which an expression evaluates whenever a run-time error occurs. Thus, the operational semantics of Definition 3.14 is extended with the error trapping rules in Figures 2.12 and 3.14.

Definition 3.20 (Extended Evaluation)**.** *Let $e \in \lambda_\chi$ and a heap S such that $(e;S)$ is a valid configuration. The evaluation relation of an expression e to a value v, written $e;S \downarrow v;S'$ is defined inductively by the rules in Figures 2.8, 2.12, 3.9, 3.10, and 3.14.*

Besides the run-time errors trapped by the rules of Figure 2.12, which correspond to λ_R expressions, we extend the operational semantics to trap the following errors: attempting to use a value, which is not a configurator, to build a component; attempting to instantiate some value which is not a component; or attempting to compose two values (using $-;-$) which are not configurators.

Some technical challenges arise in the proof of subject reduction related to typing transient states of an instance, which need to be considered during a composition. More precisely, during

(Wrong Compose)
$$\frac{e; S \downarrow v; S' \quad v \neq \mathsf{conf}(c)}{\mathsf{compose}\ e; S \downarrow \mathsf{wrong}; S}$$

(Wrong New)
$$\frac{e; S \downarrow v; S \quad v \neq \mathsf{comp}(c)}{\mathsf{new}\ e\ \mathsf{with}\ r_j := e_j\ ^{j \in 1..m}; S \downarrow \mathsf{wrong}; S}$$

(Wrong Sequence)
$$\frac{e_1; S \downarrow v; S' \quad v \neq \mathsf{conf}(c)}{(e_1; e_2); S \downarrow \mathsf{wrong}; S'}$$

(Wrong Sequence 2)
$$\frac{e_1; S \downarrow \mathsf{conf}(c_1); S' \quad e_2; S \downarrow v; S' \quad v \neq \mathsf{conf}(c)}{(e_1; e_2); S \downarrow \mathsf{wrong}; S'}$$

(Wrong Plug)
$$\frac{\text{select}_S(s, \pi_1) \text{ is undefined}}{s; \mathsf{plug}\ \pi_1\ \mathsf{into}\ \pi_2; S \Downarrow \mathsf{wrong}; S}$$

(Wrong Uses)
$$\frac{v \neq comp(c)}{s; x[v : \tau]; S \Downarrow \mathsf{wrong}; S}$$

Figure 3.14: Error trapping rules for λ_χ^τ.

the process of constructing an object, that is the need to consider architecturally inconsistent values, so we need to consider a more flexible invariant. Thus, in order to keep such global type information in the proof, we define an auxiliary type annotation, of the form $[\![\tau \Rightarrow \sigma]\!]$, to keep track, during the building process of an instance, of its unsatisfied required ports (τ) on one hand, and of the declared provided ports on the other hand (σ). The final type of the object to be built is then σ. The type τ is used just to verify the satisfaction of the required ports when a component is instantiated, or used in a composition.

Definition 3.21 (Partially Linked Object Type). *A partially linked object type is an expression of the form $[\![R \Rightarrow P]\!]$ where R and P are object types of the form $R = \{\ell_i^r : \tau_i^r\ ^{i \in 1..n}\}$ and $P = \{\ell_i^p : \tau_i^p\ ^{i \in 1..m}\}$.*

We then relate partially linked object types to objects, as follows:

Definition 3.22. *A partially linked object type of the form $[\![R \Rightarrow P]\!]$, where $R = \{\ell_i^r : \tau_i^r\ ^{i \in 1..n}\}$ and $P = \{\ell_i^p : \tau_i^p\ ^{i \in 1..m}\}$,* types an object (r, e, p) with relation to a typing environment Γ and a heap *S, if:*

- *R consists of its requirements with $\Gamma(r.\ell_i^r) = \tau_i^r \quad \forall i \in 1..n$, and*
- *P declares its provided ports with $\Gamma(p.\ell_i^p) = \tau_i^p \quad \forall i \in 1..m$.*

Notice that the partially linked object type $[\![R \Rightarrow P]\!]$ indicates not only that the object type is P but also that it depends on the services indicated in R.

We now define a notion of architectural conformance between partially built instances and resource sets, in order to specify an invariant property of the application of configurators to instances during the composition process.

Definition 3.23 (Architectural conformance). *Let $s = (r, e, p)$ be an object and Γ a typing environment such that*

- *its required ports $r = \{\ell_i^r = \imath_i^r \;^{i \in 1..n^r}\}$ are typed $\Gamma(\imath_i^r) = \tau_i^r \;^{i \in 1..n^r}$,*
- *its internal elements $e = \{\ell_i^c = \imath_i^c \;^{i \in 1..n^c}, \ell_i^m = \imath_i^m \;^{i \in 1..n^m}\}$ are typed $\Gamma(\imath_i^c) = [\![\{\ell_h^{ri} : \delta_h^i \;^{h \in 1..n_i^c}\} \Rightarrow \{\ell_h^{pi} : \gamma_h^i \;^{h \in 1..m_i^c}\}]\!] \;^{i \in 1..n^c}$ for internal component instances, and $\Gamma(\imath_i^m) = \tau_i^m \;^{i \in 1..n^m}$ for method blocks, and*
- *its provided ports $p = \{\ell_i^p = \imath_i^p \;^{i \in 1..n^p}\}$ are typed $\Gamma(\imath_i^p) = \tau_i^p \;^{i \in 1..n^p}$.*

Then, we say that s conforms with a resource list K (with relation to Γ) if

- $K_\circ \subseteq \{\ell_i^p : \tau_i^p \;^{i \in 1..n^p}\} \cup \{\ell_j^c.\ell_h^{rj} : \delta_h^j \;^{j \in 1..n^c \; h \in 1..n_j^c}\}$
- $K_\bullet \subseteq \{\ell_i^r : \tau_i^r \;^{i \in 1..n^r}\} \cup \{\ell_i^m : \tau_i^m \;^{i \in 1..n^m}\} \cup \{\ell_j^c.\ell_h^{pj} : \gamma_h^j \;^{j \in 1..n^c \; h \in 1..m_j^c}, \ell_i^c : \{\ell_h^{pi} : \gamma_h^i \;^{h \in 1..m_i^c}\}^{i \in 1..n^c}\}$
- $K_\triangleleft \subseteq \{\ell_i^r : \tau_i^r \;^{i \in 1..n^r}\}$
- $K_\triangleright \subseteq \{\ell_i^p : \tau_i^p \;^{i \in 1..n^p}\}$.

In this definition we see that: all the unsatisfied resources ($K_\circ$) must be either provided ports of the current object (ℓ_i^p) or required ports of an internal element ($\ell_j^c.\ell_h^{rj}$); all available resources ($K_\bullet$) are either required ports of the current object (ℓ_i^r), internal components (ℓ_j^c), provided ports of internal components ($\ell_j^c.\ell_h^{pj}$), or method blocks (ℓ_j^m); and, that all the expected required and provided ports must be a subset of the object's ports. (Refer to Definition 3.11 for the subset relation on interface types.) So, given a set of resources, we say that an instance conforms to it if its resources are among the instance's elements in the right category (required ports, internal elements, or provided ports), and have equivalent types. Our subject reduction ensures that on each step of the application of a configurator, the composition context conforms with the expected effect (expressed in the configurator's type).

As before, the usual properties of weakening and preservation of types under substitution of variables are essential to prove type safety.

Lemma 3.24 (Weakening). *For all typing environments Δ, Δ', all expressions $x, e \in \lambda_{\chi}^{\tau}$, and types $\tau \in \mathcal{T}_{\chi}$: If $\Delta, \Delta' \vdash e : \tau$ and $x \notin Dom(\Delta')$ then $\Delta, x : \tau', \Delta' \vdash e : \tau$.*

Proof. By induction on the height of the derivation and by case analysis of the last rule used. □

To prove the preservation of types under substitution of variables we need a partial result which asserts what kind of free names and locations may occur in component and configurator values. We prove that all configurator and component values only use names typed with component or configurator types.

Lemma 3.25. *For all typing environments Δ, values v, and types τ. If $\Delta \vdash v : \tau$ with τ a component or configurator type then $|\Delta| \vdash v : \tau$.*

Proof. According to our definition of values for λ_χ^τ (Definition 3.13), and the typing rules for values (Figures 2.11 and 3.12) there are three kinds of values that may be typed by configurator or component types. Value v is either a location, a component, or a configurator. (In practise, locations typed by configurator and component types will not result from evaluating expressions, they are nevertheless present in the heap.)

Case: $v = \imath$

By Rule (Val Location) we have that $\imath : \tau \in \Delta$, and by Definition 3.17 (Restricted typing environment) we know that $\imath : \tau \in |\Delta|$ and therefore $|\Delta| \vdash v : \tau$.

Case: $v = \mathsf{conf}(c)$

By Rule (Val Configurator Value), we have that $\Delta \vdash c : K \Longrightarrow K'$. We then have to analyse the cases for composition operation c.

Subcase: $\mathsf{provides}\ \ell : \sigma$, $\mathsf{requires}\ \ell : \sigma$, **and** $\mathsf{plug}\ \pi_1 : \tau_1\ \mathsf{into}\ \pi_2 : \tau_2$

The typing of these expressions does not depend on the typing environment. For instance, if $c = \mathsf{provides}\ \ell : \sigma$ then the type is the same for all typing environments $|\Delta| \vdash (\mathsf{provides}\ \ell : \sigma) : \tau$. The same reasoning applies to the other two expressions in this case.

Subcase: $x[v : \tau \Rightarrow \sigma]$

In this case, by Rule (Comp Uses), we know that $\Delta \vdash v : \tau \Rightarrow \sigma$. By induction hypothesis we have that $|\Delta| \vdash v : \tau \Rightarrow \sigma$ and therefore $|\Delta| \vdash x[v : \tau \Rightarrow \sigma] : \tau$.

Subcase: $x_I[\ell_i : \tau_i = e_i\ {}^{i \in 1..n}]$

We have that, $|\Delta|, \ell'_i : \tau'_i\ {}^{i \in 1..m}, x : \{|\ell_i : \tau_i\ {}^{i \in 1..n}|\} \vdash e_i : \tau_i$ for all $i \in 1..n$ which immediately supports $|\Delta| \vdash x_I[\ell_i : \tau_i = e_i\ {}^{i \in 1..n}] : \tau$ by Rule (Comp Uses).

Subcase: $c_1; c_2$

By induction hypothesis on the premises of Rule (Comp Sequence).

Case: $v = \mathsf{comp}(c)$

We have $\Delta \vdash \mathsf{compose}\ c : \tau$ which implies $\Delta \vdash c : K \Longrightarrow K'$ such that $\tau = K_\triangleleft \Longrightarrow K_\triangleright$. The cases to be analysed for the composition operation c are the same as in the previous case and lead to the conclusion that $|\Delta| \vdash v : \tau$. □

The property that component and configurator values are stateless values follows from Lemma 3.25 in conjunction with a property about the representation of state. Remember that we define the notion of state variables by means of (mutable) records (by disallowing free manipulation of locations). Thus, stateful values resulting from evaluating language expressions must refer to the heap through a record, i.e. locations occurring in such values must be typed by record or interface types.

Definition 3.26 (record-based expressions)**.** *For all expressions e and typing environments Γ we say that e is* record-based with relation to Γ *if for all $\imath \in FL(e)$, $\Gamma \vdash \imath : \tau$ with τ a record or interface type.*

Moreover, this property on expressions, and in particular on values, must be preserved by the operations that manipulate state, e.g. select and assign. We define a property on heaps that defines this. All records in a heap (state variables) are assigned either stateless values (with no locations) or stateful values whose locations lead to records.

Definition 3.27 (record-based heaps)**.** *For all heaps S and typing environments Γ typing S we say that S is* record-based with relation to Γ *if for all $\imath \in \mathrm{Dom}(S)$ such that $S(\imath) = \{\ell_i = \imath_i{}^{i \in 1..n}\}$ we have $S(\imath_i) = \mathsf{nil}$ or $S(\imath_i)$ is record-based with relation to Γ.*

We now prove the preservation of types under substitution. We simultaneously assert the preservation of the record-based property on expressions.

Lemma 3.28 (Substitution)**.** *For all typing environments Δ, Δ', all expressions $x, e \in \lambda_\chi^\tau$, and types $\tau \in \mathcal{T}_\chi$: If $\Delta, x : \tau, \Delta' \vdash e : \tau'$ and $\Delta \vdash v : \tau$ then $\Delta, \Delta' \vdash e[x \leftarrow v] : \tau'$. Moreover, if e and v are record-based then $e[x \leftarrow v]$ is also record-based.*

Proof. By induction on the height of the derivation and by case analysis of the last rule used. We here address the most interesting cases, all the others follow by simple induction on the premises of the corresponding rules.

Case: **(Val Var)**

We have that $\Delta, x : \tau, \Delta' \vdash y : \sigma$ and $\Delta \vdash v : \tau$. There are two possible cases, either the variable is replaced or not. If $x = y$ then we have that $\tau = \sigma$ and that, by Lemma 3.24 (weakening) we obtain that $\Delta, \Delta' \vdash v : \tau$ and if the locations occurring in v are only typed using record and interface types they are the only ones in the resulting expression. If the variables are the same then the substitution is not performed. The resulting expression is y and there are no locations in y to be typed.

Case: **(Val Location)**

We have that $\Delta, x : \tau, \Delta' \vdash \imath : \sigma$ because $\imath : \sigma \in \Delta, x : \tau, \Delta'$ and therefore $\imath : \sigma \in \Delta, \Delta'$ and thus $\Delta, \Delta' \vdash \imath : \sigma$ and x does not get replaced in $\imath$. If the location is typed by a record or interface type then its typing is kept in the conditions of the lemma.

Case: **(Val Abstraction)**

In this case we have that $e = \lambda y : \sigma.e'$. Assuming that the replaced variable is not captured by the abstraction we have, $\Delta, x : \tau, \Delta', y : \sigma \vdash e : \sigma'$. By induction hypothesis, we have that $\Delta, \Delta', y : \sigma \vdash e[x \leftarrow v] : \sigma'$ and therefore $\Delta, \Delta' \vdash (\lambda y : \sigma.e')[x \leftarrow v] : \sigma \rightarrow \sigma'$. Observe that the locations in the resulting expression are typed by record and interface types if they are typed this way in the original expression e and value v.

***Subcase:* (Comp Provides), (Comp Requires), and (Comp Plug)**

The typing of the operations is the same no matter what the typing environment declares. Thus, the lemma holds.

***Case:* (Comp Uses)**

We now that $\Delta, x : \tau, \Delta' \vdash y[e : \tau \Rightarrow \sigma] : \emptyset \Longrightarrow K$ that is supported by $\Delta, x : \tau, \Delta' \vdash e : \tau \Rightarrow \sigma$. By induction hypothesis we obtain $\Delta, \Delta' \vdash e[x \leftarrow v] : \tau \Rightarrow \sigma$ and therefore $\Delta, \Delta' \vdash y[e : \tau \Rightarrow \sigma][x \leftarrow v]$. The properties on the locations occurring in the value v and expression e hold in $e[x \leftarrow v]$.

***Case:* (Comp Method Block)**

If $e = y_I[\ell_i : \tau_i = e_i \ ^{i \in 1..n}]$ we have $\Delta, x : \tau, \Delta' \vdash y_I[\ell_i : \tau_i = e_i \ ^{i \in 1..n}] : K \Longrightarrow K, \{y \bullet \{\ell_i : \tau_i \ ^{i \in 1..n}\}\}$. By Rule (Comp Method Block) we have $|\Delta, x : \tau, \Delta'|, \ell'_i : \tau'_i \ ^{i \in 1..m}, y : \{\!|\ell_i : \tau_i \ ^{i \in 1..n}|\!\} \vdash e_i : \tau_i$ for all $i \in 1..n$.

Then, we have two possible cases for the shape of τ. Either type τ is a component or configurator type, or it is not.

***Subcase:* $\tau = R \Rightarrow P$ or $\tau = K \Longrightarrow K'$**

In this case the type assignment $x : \tau$ persists in the typing environment of the premise. $|\Delta|, x : \tau, |\Delta'|, \ell'_i : \tau'_i \ ^{i \in 1..m}, y : \{\!|\ell_i : \tau_i \ ^{i \in 1..n}|\!\} \vdash e_i : \tau_i$. Notice that the names ℓ_i and y operation are bound in the expressions e_i. This is visible in Definition 3.3. By induction hypothesis we have that $|\Delta, \Delta'|, \ell'_i : \tau'_i \ ^{i \in 1..m}, y : \{\!|\ell_i : \tau_i \ ^{i \in 1..n}|\!\} \vdash e_i[x \leftarrow v] : \tau_i$ in the case x is not captured by either the method block name or the imported labels, or $|\Delta, \Delta'|, \ell'_i : \tau'_i \ ^{i \in 1..m}, y : \{\!|\ell_i : \tau_i \ ^{i \in 1..n}|\!\} \vdash e_i : \tau_i$ if it is captured. We then have that $\Delta, \Delta' \vdash y_I[\ell_i : \tau_i = e_i \ ^{i \in 1..n}][x \leftarrow v] : K \Longrightarrow K, \{y \bullet \{\ell_i : \tau_i \ ^{i \in 1..n}\}\}$.

***Subcase:* τ has another shape**

If τ is not a component or configurator type then $|\Delta, x : \tau, \Delta'| = |\Delta, \Delta'|$ and the result follows naturally. □

We now enunciate a general lemma which asserts subject reduction considering simultaneously the evaluation of expressions and the application of composition operations. The lemma brings explicit the invariants of type preservation and absence of nil references in the evaluation judgements on well-typed expressions. In the configurator application evaluation judgement, we prove an invariant that implies the conformance between the configurator type and the structure of the instance. It also ensures that the nil references that exist in the heap correspond to uninitialised ports of instances. In the cases where the evaluation judgement uses the application judgement, these nil values get eliminated from the heap (cf. the plug-assignments in the new e expression).

Lemma 3.29 (Subject Reduction)**.**

1. *Let $(e; S)$ be a valid configuration in $\lambda_\chi^\tau \backslash \{\mathsf{nil}\}$ such that $\mathsf{nil}(S) = \emptyset$ and let Γ be a typing environment typing S such that e and S are record-based with relation to Γ.*

   If $\Gamma \vdash e : \tau$ and $(e; S \downarrow v; S')$ then:

   a) there is a Γ' that extends Γ and types S',

   b) $\Gamma' \vdash v : \tau$,

   c) v is either an abstraction, a component, a configurator, or a location that is either `undefined` or `refers-to` a record,

   d) v and S' are record-based with relation to Γ, and

   e) $\mathsf{nil}(S') = \emptyset$.

2. *Let c be an expression such that $\Gamma \vdash c : K \Longrightarrow K'$, let S be a heap such that, for some set $X \in Dom(S)$, $\mathsf{nil}(S) \subseteq \{\mathrm{select}_S(s, \pi) \mid (\pi : \tau) \in K_\triangleleft \cup K_\circ\} \uplus X$ and Γ types S.*

   Let s be a partially linked object s such that it conforms with K and its partially linked object type is $[\![R \oplus K_\triangleleft \Rightarrow P \oplus K_\triangleright]\!]$ and s and S are record-based with relation to Γ:

   If $s; c; S \Downarrow s'; S'$ then

   a) there is Γ' typing S' and extending Γ,

   b) s' is a partially linked object that extends s and conforms with K'. Its partially linked object type is $[\![R \oplus K'_\triangleleft \Rightarrow P \oplus K'_\triangleright]\!]$,

   c) s' and S' are record-based with relation to Γ', and

   d) $\mathsf{nil}(S') \subseteq \{\mathrm{select}_{S'}(s, \pi) \mid (\pi : \tau) \in K'_\triangleleft \cup K'_\circ\} \uplus X$.

Proof Sketch. For the sake of legibility we only show a sketch of the proof. The complete proof can be found in appendix A on page 176. The proof is carried out by induction in both cases of the lemma. On the first case, we prove it by induction on the size of the evaluation derivations and by analysis of the last rule used. We use the second case when necessary.

In the cases where the last rule in the derivation is one of the rules of λ_R^τ ((Eval Value), (Eval Application), (Eval Record), (Eval Assign), and (Eval Select)), and also in the case where the last rule is (Eval Compose) the proof is obtained by direct application of the induction hypothesis and using Definition 2.21 to assert the type of the values in the heap. In the case of Rule (Eval New), the proof is as follows:

***Case:* (Eval New)**

The hypothesis $\Gamma \vdash \mathsf{new}\ e\ \mathsf{with}\ \ell^r_j := e_j\ {}^{j\in 1..n} : \{\ell^p_i : \sigma_i\ {}^{i\in 1..n}\}$ is obtained by Rule (Val New) with the premises $\Gamma \vdash e : \tau$ with $\tau = \{\ell^r_j : \tau_j\ {}^{j\in 1..m}\} \Rightarrow \{\ell^p_i : \sigma_i\ {}^{i\in 1..n}\}$. new is evaluated by Rule (Eval New), thus, if $\mathsf{new}\ e\ \mathsf{with}\ \ell^r_j := e_j\ {}^{j\in 1..m}; S \downarrow (r, e, p); S'$, then we must have $e; S \downarrow \mathsf{comp}(c); S_0$. By induction hypothesis we have that there is a Γ_0 extending Γ such that $\Gamma_0 \vdash \mathsf{comp}(c) : \tau$ and $\mathsf{nil}(S_0) = \emptyset$.

Now, notice that the order of the premises does not reflect the real dependence between them. For technical reasons, the plug-assignments must be considered first in this proof. By iteratively applying Lemma 3.28 (substitution), the induction hypothesis, and Lemma 3.24 (weakening) on the typing and evaluation judgements of each plug-assignment expression, $(\Gamma \vdash e_i : \tau_i)$ and $(e_i; S_{i-1} \downarrow v_i; S_i)$, we obtain that for all $i \in 1..n$ we have Γ_i extending Γ_{i-1} and typing S_i such that $\Gamma_i \vdash v_i : \tau_i$ and $\mathsf{nil}(S_i) = \emptyset$.

By Lemma 3.24 (weakening) on the typing of $\mathsf{comp}(c)$ and Rule (Val Composition Value) we have $\Gamma_n \vdash c : \emptyset \Longrightarrow K$ and $K_\circ = \emptyset$, with type $\tau = K_\triangleleft \Rightarrow K_\triangleright$ thus $K_\triangleleft = \{\ell_j : \tau_j\ {}^{j\in 1..m}\}$. Notice that the judgement above applies c to the empty instance $(\mathbf{0}; c; S_n \Downarrow s; S_{n+1})$, and that $\mathsf{nil}(S_n) = \emptyset$ meets the conditions of the second part of the lemma with $X = \emptyset$. In this case, $\mathbf{0}$ conforms with the resource list $\emptyset$ and has type $[\![\{\} \Rightarrow \{\}]\!]$. By induction hypothesis on the second case of the lemma, we have that there is a Γ_{n+1} extending Γ_n and the resulting instance s conforms with K with the partial type $[\![K_\triangleleft \Rightarrow K_\triangleright]\!]$. We know that the locations of the required ports are $\{\,l_i\ {}^{i\in 1..n}\} = \{\text{select}_{S'}(s, \ell) \mid (\ell : \tau) \in K_\triangleleft\}$ and that $\mathsf{nil}(S') \subseteq \{\text{select}_{S'}(s, \ell) \mid (\ell : \tau) \in K_\triangleleft\}$.

Γ_{n+1} types S_{n+1} and the locations in $s_\triangleleft = \{\ell_i \mapsto l_i\ {}^{i\in 1..n}\}$ are in S_n. From the conformance of s with K and Rule (Val Record) we know that $\Gamma_{n+1} \vdash s : \{\!|\ldots, \ell^p_j : \sigma_j\ {}^{j\in 1..m}|\!\}$ where $K_\triangleright = \{\ell^p_j : \sigma_j\ {}^{j\in 1..m}\}$ and by Rule (Val Interface), we have a $\Gamma' = \Gamma_{n+1}, l : K_\triangleright$, extending Γ, and typing $S_{n+1}[\,l \mapsto s][\,l_i \mapsto v_i\ {}^{i\in 1..n}]$. We finally have that l is a location that *refers-to* an object, which is a record, such that $\Gamma' \vdash l : K_\triangleright$ with $K_\triangleright = \{\ell^p_j : \sigma_j\ {}^{j\in 1..m}\}$. Finally, since $v_i \neq \mathsf{nil}$ for all $i \in 1..n$ we have that $\mathsf{nil}(S^{n+1}[\,l \mapsto s][\,l_j \mapsto v_j\ {}^{j\in 1..m}]) = \emptyset$ with s being the newly created instance. Observe that d) holds.

For the composition expressions, when evaluated in a computational context, the proof follows by direct application of Rule (Val Configurator Value). An exception is made for the operations with internal expressions (the introduction of an internal component) whose proof is as follows:

***Case:* (App Uses)**

The expression $x[v : \tau \Rightarrow \sigma]$ where $\tau = \{\ell^r_i : \tau_i\ {}^{i\in 1..n}\}$ and $\sigma = \{\ell^p_j : \sigma_j\ {}^{j\in 1..m}\}$ is typed by a judgement $\Gamma \vdash x[v : \tau \Rightarrow \sigma] : \emptyset \Longrightarrow K$ with $K = \{x \bullet \sigma, x.\ell^r_i \circ \tau'_i{}^{i \in 1..n}, x.\ell^p_j \bullet \sigma'_j{}^{j \in 1..m}\}$. By Rule (Comp Uses) we have that $\Gamma \vdash v : \tau \Rightarrow \sigma$. Rule (Val Composition Value) is the only one that types a component value and therefore we have $v = \mathsf{comp}(c)$ with $\Gamma \vdash c : \emptyset \Longrightarrow K'$ with $K'_{\circ} = \emptyset$ and $(\tau \Rightarrow \sigma) = (K'_{\triangleleft} \Rightarrow K'_{\triangleright})$.

From the application judgement, $s; x[v : \tau \Rightarrow \sigma]; S \Downarrow s'; S'$, we know that $(\mathsf{new}\ v); S \downarrow \imath; S'$. Since v only evaluates to itself with no changes to the heap, and there are no plug-assignments, the evaluation of this **new** expression depends solely on $\mathbf{0}; c; S \Downarrow s; S''$ with $S' = S''[\imath \mapsto s]$.

Since $\Gamma \vdash x[v : \tau \Rightarrow \sigma] : \emptyset \Longrightarrow K$, where the set of demanded resources is empty, we know that there is a set X such that $\mathsf{nil}(S) \subseteq X$. By induction hypothesis on the height of the evaluation derivation we have that there is a Γ' that types S'' and that the resulting value $s = (r', e', p')$ conforms with K' with the partially linked object type $[\![K'_{\triangleleft} \Longrightarrow K'_{\triangleright}]\!]$ with relation to Γ''. We know that $s' = (r, e \oplus \{x \mapsto \imath\}, p)$ and the store S' are typed by $\Gamma'' = \Gamma', \imath : K'_{\triangleright}$. Notice that the instance type corresponding to $[\![\tau \Rightarrow \sigma]\!]$ is $K'_{\triangleright}$. More, the instance s' conforms, according to Definition 3.23, with the resource set K. By the induction hypothesis we also know that $\mathsf{nil}(S'') \subseteq \{\text{select}_{S''}(s, \ell) \mid (\ell : \tau) \in K'_{\triangleleft}\} \cup X$ with $K'_{\triangleleft} = \{\ell^r_i : \tau_i{}^{i \in 1..n}\}$. We then conclude that, in the context of the containing instance, $\mathsf{nil}(S') \subseteq \{\text{select}_{S''}(s', \pi) \mid (\pi : \tau) \in \{x.\ell^r_i : \tau_i{}^{i \in 1..n}\}\} \cup X$. The partial type of the resulting object remains unchanged since no required or provided ports are added to the instance.

To complete the proof, with a reasoning analogous to the one used in Theorem 2.24, we prove that a well-typed expression never evaluates to the value wrong and consequently that there are no run-time errors due to nil values. (for the complete proof see appendix A on page 176). □

Notice that Lemma 3.29 asserts that the relevant part of the heap which is still "unlinked" ($\mathsf{nil}(S')$) is appropriately characterised by the set K' mentioned in the type of the configurator. In particular, if $K'_{\circ}$ is empty as is the case in the type of an instantiable component, only the required ports of the generated instance still need to be satisfied.

The main result of this chapter then follows as a corollary of lemma 3.29. The subject reduction theorem yields type safety for the language λ^{τ}_{χ} as follows:

Theorem 3.30 (Subject Reduction)**.** *Let $(e; S)$ be a valid configuration in $\lambda^{\tau}_{\chi}\backslash\{\mathsf{nil}\}$ such that $\mathsf{nil}(S) = \emptyset$ and let Γ be a typing environment typing S such that e and S are record-based with relation to Γ. If $\Gamma \vdash e : \tau$ and $e; S \downarrow v; S'$ then*

a) there is a Γ' that extends Γ and types S',

b) $\Gamma' \vdash v : \tau$,

c) v is either an abstraction, a component, a configurator, or a location that is either `undefined` or `refers-to` a record,

d) If v is a component or configurator value then $FL(v) = \emptyset$, and

e) $\mathsf{nil}(S') = \emptyset$.

Proof. This theorem results directly from the first case of Lemma 3.29. In d) we use Lemma 3.25 that states that, for configurator and component values, $\Gamma \vdash v : \tau$ implies that $|\Gamma| \vdash v : \tau$ which, together with the result of Lemma 3.29 d), out rules the possibility of any locations occurring in v. Remember that locations typed as record and interface types declared in Γ, which are the only ones that may occur in the language results, are eliminated in $|\Gamma|$ and therefore may not occur in v. □

Notice that architectural soundness, is ensured by Lemma 3.29, and consequently by Theorem 3.30. Remember the definition of the typing of heaps (Definition 2.21) which ensures that, in well-typed heaps, all locations either lead to values of the expected type, lead to nil, or are *undefined*. Thus, in a well-typed heap with no nil values, as it is the case of the heaps resulting from evaluating a well-typed expression, all ports of instances, which are locations in the heap, are linked. Ports may be linked to a value of the expected type, i.e. a record containing the expected set of methods, or they may be *undefined*, which means that any call to such a port would not terminate. These cycles are only introduced if explicitly programmed. A quick analysis of the invariant imposed in the second part of Lemma 3.29 shows that the nil values introduced during the construction of an instance, are in the end replaced by meaningful values. This is particularly visible in the cases (Eval New) and (App Uses) of the type safety proof above.

Notice also that Theorem 3.30 d) makes precise the claim that configurators and components, resulting from evaluating well-typed programs, are pure stateless values (independent of the heap).

3.3 Remarks

In this chapter we have defined a core component calculus that follows the design principles enunciated in the Introduction. Its first-class values consisting of configurators, components and objects and its operations are designed so that the structures of components can be defined dynamically and type safely. We now discuss some issues about the language that we think are worth noticing.

As already mentioned in Section 3.1.1, our semantics provides a clear notion of phase distinction. Our evaluation relation imposes a separation between the computation of the structures of components and configurators and its actual assembly (which in our case happens at instantiation-time). In particular, the computation of the effect of configurators (composition operations fully evaluated) only manipulates components and configurators, and terminates in all cases. Our type system reinforces this separation by preventing run-time errors such as the attempting of using a value which is not a component as an internal component in a composition.

Furthermore, the type system enforces type safety, which in our setting implies, not only the absence of null dereferencing errors and "method not implemented" errors, but also the architectural consistency of dynamic composition processes, i.e. the absence of "service not implemented" errors, which are fairly common run-time errors found in component-oriented programming.

Another issue ensured by our type system concerns the stateless character of component and configurator values. This feature is important in a setting of component-based software development where it is common the reuse of third-party components, either stored in component libraries or transmitted across the Internet.

One issue that is also important mentioned here are the design options of our language λ_{χ}^{τ}. In particular, the set of composition operations in our language, which seems to capture the definition of components providing and requiring services and built by aggregation and adaptation of other components. The basic operations to construct a component: two operations that declare required (imported) and provided (implemented) services (requires and provides), one operation that introduces an internal component ($x[-]$), and one operation that connects two ports (plug), and therefore binds a concrete implementation of a service to a declared need for that service.

These operations only allow the structuring of components from other components. It is not possible, using these operations, to define the base functionality of a component, nor to adapt the functionality of an internal component. This can be obtained either by allowing some form of base component declaration, or the more flexible approach we have chosen. Our approach is to define another kind of architectural block, method blocks ($x_{-}[- = -]$). Method blocks implement services and have free access to the elements of the component structures they are inserted in. They define, in-place, the implementation and adaptation of services. We finally use an operation for composing two configurators. We have chosen what it seems to be the minimal set of operations that allows the definition of component structures, in a declarative style, and at the same time allows the typing of such constructions in such a way that architectural soundness can be ensured statically.

Additionally, the design of some expressions was influenced by the design of the type system. In particular, the type annotations in expressions such as the expressions for the introduction of method block ($x_I[- = -]$) and the plug expressions ($\mathsf{plug}\ \pi : \tau\ \mathsf{into}\ \pi : \tau$) could have been omitted at the expense of defining some type inference mechanism. We chose to decorate the expressions with extra type expressions and keep the type system design simpler. The type annotation in the expression for the introduction of internal components is already present in the syntax of λ_χ^τ for the sake of uniformity with languages defined in subsequent chapters. In particular, it is essential when dealing with subsumption of component values.

Finally, notice that Example 3.8 defined in the untyped calculus λ_χ can be trivially extended to the typed calculus λ_χ^τ, thus λ_χ^τ encodes object-oriented mechanisms such as implementation inheritance and mixin application in a typeful way.

Now, that we have presented our programming language with dynamic composition and configuration of components, we compare and contrast our approach with that of other authors.

3.4 Related Work

At first glance, the language design of λ_χ resembles that of an architecture description language like Darwin [65, 66, 71, 72]. Although the composition operations of our calculus may have similar interpretations in ADLs, the application domain and the techniques used are quite different. ADLs target the specification and verification, at a high level of abstraction, of coordination levels for distributed systems. We on the other hand, use the notion of architecture to structure programs, potentiate separate development, and maximise reuse of code at a more detailed level, the programming language level. Unlike the ADLs, our approach targets the development of tools for static checking the actual code, in particular of type systems, instead of generating coordination code from a high level specification to be completed afterwards with computational code.

A more profound insight relates our work more closely to those on programming languages and in particular with languages providing sophisticated module mechanisms [16, 44, 101, 11, 56, 57] or with object-oriented language reuse mechanisms like mixins [16, 15, 43] and traits [80, 91]. Among others we distinguish the earlier works of Bracha [17] which adds the use of well defined operators on modules, and that of Flatt and Felleisen [44] which introduces mutually recursive and first-class module construction at the programming language level. In this section, we compare our work in more detail with the approaches of Flatt et al. [44, 74, 79] and that of Ancona and Zucca [11], a calculus derived from the initial concepts of Bracha's Jigsaw language [16]. We next relate our work with other component based languages, which, to the best of our knowledge, followed an approach similar to ours but were developed with slight different focuses [4, 92, 103].

Units and Jiazzi The work of Flatt and Felleisen [44, 43] introduces the modularity mechanisms at the programming language level. In their approach, basic module entities, called *Units*, are first-class sets of declarations from which some names may be exported and where imported names can occur; each Unit also has an initialisation expression. They define programming language abstractions capable of expressing the creation and invocation of a Unit, thus evaluating the initialisation expression, and the creation of *Compound Units*, based on other existing units. Compound units can only be used as values once rewritten to a ground flat unit expression.

Some similarities and differences between our calculus, λ_χ^τ, and the language of Units are worth mentioning here. First, instead of being a means of producing a complete program source from separate declarations, components in λ_χ^τ assume a role of generating entities for objects; Although the structure of an object is flattened in terms of memory references, the hierarchy of name bindings defined in the structure of components remains intact within their instances. This ensures a much simpler instantiation of compound components (without the need for renaming operations) and allows, in a later stage, for reconfiguration actions to be defined in terms of the structure of the originating component (Chapter 4); It also allows for the elements of an object's structure to be treated as black-boxes.

One feature that is not covered by our component language is the Units' import and export mechanism for types. Apart from the definition of abstract types, the unification of type variables between internal components can be coded in our language with parametric polymorphism which we present ahead in Chapter 6.

Our basic structure of component types resembles that of units. However, we support typing of composition and instantiation operations using a single type form for configurators which does not seem to be doable with the types of Units. Although we can extend Units with imperative object-oriented language constructs, the composition of object producing units does not extend to compound units which produce the composition of their element's instances. Unlike this, λ_χ^τ provides primitive structuring mechanisms which are uniform in both components and objects. When a λ_χ^τ component is instantiated, the instances of its internal elements are interconnected in a stateful way, i.e. within the state of each instance, and a wrapping instance is created. To emulate our instantiation process one would have to add some custom "constructor Units" to produce the wrapping instances. For the purpose of composition, the required and provided ports can be made in the way of Units' imports and exports. Another necessary add-on to Units is some kind of renaming mechanisms for import and export names, which already appears in some extent in a tag mechanism in a subsequent work [43].

In special cases of λ_χ expressions, as it is the case of the language presented in [81], it is possible to find an encoding of λ_χ operations in the language of Units to produce similar objects: (Consider the same base language with functions and records)

```
c = compose (provides p;
             m[f=fun x → x+1];
             plug m into p)
```

```
M = unit export m
      val m = {f = fun x → x+1} in m

P = unit import m export p
     val p = m in p

Ctor = unit import p in {p=p}

c = compound export p
     link M provides m
     and P with m provides p
     and Ctor with p
```

We defined a unit for each composition operation in the component declaration. Notice that the names in the example were chosen to avoid conflicts because, in the language of units, they are connected by capturing the names in the context of the compound. The last element of the compound (Ctor) is here introduced to export, in an object (encoded as a mutable record), all the exported ports. A component with required ports and its composition with c above can be encoded in the following way:

```
d = compose(requires p;
            provides q;
            m[f=fun x → p.f(x)];
            plug m into q)
e = compose(provides q;
            provides p;
            c[c]; d[d];
            plug c.p into d.p;
            plug d.q into q;
            plug c.p into p)
```

```
M2 = unit import p export m
   val m = {f=fun x → p.f(x)} in m
Q2 = unit import m export q
   val q = m in q
Ctor2 = unit import q in {q=q}

d = compound import p export q
   link M2 provides m
   and Q2 with m provides q
   and Ctor2 with q

Ctor3 = unit import p,q in {p=p,q=q}

e = compound export p,q
    link c provides p
    and d with p provides q
    and con with p,q
```

Although this translation is type preserving with relation to the type system presented in [81], it is not clear that the same thing can be made to λ_χ^τ expressions, namely with relation to the composition of configurators.

The opposite encoding is also possible with no real restrictions and following a simple convention regarding the resulting value, making it available at port o. A Unit can be encoded by a single component as follows:

```
unit import a export b
   val b = e
   in o
```

```
compose (requires a;provides b;provides o;
         methods m {b=e[m.b/b], o=o[m.b/b]};
         plug m into b;
         plug m into o )
```

Notice that we require the name of the method block to be used in the method bodies and therefore proceed to an explicit substitution. A compound Unit can also be encoded by a component as follows:

```
compound import q export r
         link u1 with q,r provides p
         and u2 with p provides r
```

```
compose (requires q;provides r;provides o;
         u1[u1];
         u2[u2];
         plug q into u1.q;
         plug u1.p into u2.p;
         plug u2.r into u1.r;
         plug u2.r into r;
         provides u2.o into o)
```

Notice that we explicitly link the names in λ_χ^τ in opposition to the implicit name resolution in compound units. Notice also the convention of providing o for the initialiser expression.

In conclusion, the main difference to the language of Units is that in our case, we treat the operations as first-class values instead of the whole components. The type information that we use to characterise configurators is not encodable in the types of Units.

Jiazzi [74] is the application of the notion of Units to an object-oriented like Java. Jiazzi defines blueprints, which are groups of class declarations with imported and exported class names. It allows for class usage across component boundaries (including inheritance) and the definition of mutually recursive class structures. However, all constructions are resolved at compilation-time. In opposition to this, our calculus and in particular componentJ (The concrete language derived from λ_χ and presented in Section 6.6), defines components as first-class and provide tools for developing applications without the usage of class abstraction and implementation inheritance. Although composition is at the programming language level we leave implementation inheritance to be used inside native Java components, typically in particular situations of fine-grain programming where it is more useful. Jiazzi also provides a primitive mixin mechanisms for Java, which is also proposed by Jam [7]. Although we can encode this using our language (as shown in Example 3.8) it is not present in componentJ.

Module calculi The initial study of Bracha [16, 17] on inheritance mechanisms in object-oriented languages introduced a notion of module, resulting from the simplification of the concept of a class, and a set of operators on modules which can be used to describe the usual object-oriented constructions. Then, a series of calculus emerged, using the so-called *mixin-modules* as basic structure [38, 11, 101, 56, 57, 41], which strongly resembles our initial notion of component [81]. All agree on the definition of a module as set of mutually recursive definitions with output components (distinguished by a name) whose implementation may be defined in a core language or based on deferred input components.

To illustrate these module calculi we take the calculus of Ancona and Zucca, CMS [11]. It allows the creation of modules from basic components defined in a core language. Evaluation of module and core expressions happen in different levels. CMS operations allow the merging of two modules, and the renaming and hiding of existing names. As in the general case, renaming may link module's components by capturing the input component names and replacing them by output component names. As we show next, it is possible, by merging modules, rewriting and hiding names, to combine modules in an interesting way.

We now intuitively illustrate the usage of CMS by adapting an example from [41]. Capital letters are used for component names, lowercase letters are used for variables. For the sake of simplicity, we use integer literals and operations to illustrate computations in the core language. A module is written $[\iota; o; \rho]$ where ι is a mapping from variables to component names (not bound to any implementation), o maps component names (visible in the outer context) to expressions and ρ maps variables to expressions. The scope of the declared variables is the module expression itself. So, ι declares input components, o defines the exported components of the module and ρ defines its local components (expressions of the core language).

Consider the module expression e_1 defined by,

$e_1 \triangleq [; X \mapsto 1 + 2;]$

with no deferred input component names and no local components. X is an output component denoting the core expression $1 + 2$. Then, consider a module e_2 defined by:

$e_2 \triangleq [x \mapsto Z; Y \mapsto x + y; y \mapsto 2]$

where x maps to an outer component Z and the definition of Y depends on both the input component linked to x and a local component y. One basic operation on modules is **sum**, which merges two modules together. Applied to the modules defined above, we write $e_3 \triangleq e_1 + e_2$ to obtain the result:

$e_3 = [x \mapsto Z;\ Y \mapsto x + y, X \mapsto 1 + 2;\ y \mapsto 2]$.

Although not visible in this example, a precondition to **sum** is that the two sets of names of output components are disjoint. α-renaming of local component names is performed to avoid conflicts of names and common input component names are merged together. Another operation is **freeze** which renames input components names capturing output component names. For instance, the expression $e_4 \triangleq \mathrm{freeze}_{Z \mapsto X}(e_3)$ results in:

$e_4 = [;Y \mapsto x + y, X \mapsto 1 + 2;\ y \mapsto 2, x \mapsto 1 + 2]$

In this case, the input component Z was resolved using the implementation of X. A third operation is **reduct** which manipulates the output and input component names. For instance $e_5 \triangleq_{\_ \mapsto W|}\, e_{4|Y \mapsto V}$ results in

$e_5 = [w \mapsto W; V \mapsto x + y;\ y \mapsto 2, x \mapsto 1 + 2]$

where w is fresh, Y is renamed to V and X is eliminated. CMS then provides a selection primitive to extract a component to be executed. $e_5{\downarrow}_V$ reduces to $1 + 2 + 2$. Free variables of the core expression are closed by using the substitutions available within the module.

A typed version of CMS is presented in [11] which relates with the basic type system we presented in [81]. It ensures, as we do, that a module is only used once its implementation is complete, i.e. it has no pending input components. However, the basic building block is the module construction as a whole. Our types for composition expressions, on the other hand, capture more than the extensional information of input and output components. An analogy with the operations of CMS, it would be as if **sum**, **freeze**, **reduct** were first-class values which could be freely combined at run-time. Similarly to Wells and Vestergaard's *m-calculus* [101] and more recently in Fagorzi and Zucca's *R-calculus* [41, 42], we also treat our module entities (components) as first-class values.

The composition mechanism used in CMS (**sum**), can be seen as "black-box" because it only manipulates the output and input components of modules. Local component names get α-renamed in **sum** operations so that name conflicts are avoided and that the implementation of output components is not changed. We, on the other hand, introduce "white-box" composition of basic composition operations (configurators) and "black-box" composition of closed components values. We use an operation, compose, that abstracts the implementation details of architectures which are well-formed. We ensure by typing that the results of such operation are closed component values. Hence, we know that all component instances are structurally well-formed independently of their internal structure.

Furthermore, by presenting a calculus that integrates an imperative language with composition operations, we explore the interactions between computational and meta-level operations on component architectures and object structures, notably we explore the influence of computation and state on dynamic composition of new components and, further ahead, of reconfiguration of running instances.

Component based languages ArchJava [4] and ACOEL [92] are two Java like languages which also define composition as a structuring mechanism at the programming language level. These languages support some composition constructs and allow for the verification of architectures, however they do not provide a clear separation between adaptation and composition code in programs.

A component in ArchJava is an instance of a so-called *component class*, which, besides holding state variables and implementing methods like a regular class, defines communication ports with declared provided and required methods. Component classes can then be included in the definition of other component classes (*composite components*), where their ports are explicitly connected. Internal components are used as regular instance variables in the methods of the enclosing component class.

ArchJava allows for dynamic construction of structures within pre-established connection patterns. We, on the other hand, treat components as first-class values thus allowing computations over the structure of programs and a verification based on types. Although scripting and composition are expressed at the same level, ArchJava does not present any mechanism for satisfying requirements of internal components via the component class itself. In our calculus it is possible to connect local methods to a internal component's required port and, in this way, configure its behaviour. It is also possible, in our calculus to export a internal component's functionality, by directly connection of its ports, which seems to be impossible in ArchJava.

ACOEL [92] defines components as instantiable entities which can import external class and export internal class implementations which are explicitly connected to output ports. Internal classes can be directly defined inside the component declaration or by mixin application. We, on the other hand, do not use any inheritance or extension mechanism other than composition.

Another work using composition at the programming language level is Zenger's component calculus [103], built as an extension of Featherweight Java [59]. Unlike us, their approach is driven by an operational view of evolution and extension of components rather than taking a declarative approach to architecture definitions.

Traits and mixins in object-oriented languages Mixins [16, 15] and traits [80, 91] are forms of reuse that closely relate to composition. Both provide loose coupling between the base functionality and the resulting extensions by means of well-defined expected interfaces and an application operation independent from the definitions of classes, mixins, or traits. Traits differ from the notion of mixin because they only carry code and provide a more flexible composition mechanism, e.g. in Chai [91] more than one trait can be applied to a class in a single extension operation and they can be applied to other traits to produce richer traits. This composition works independently of the standard inheritance mechanism. Mixins, on the other hand, are supported in the inheritance mechanism of object-oriented languages and therefore are less flexible. However, they are able to declare new instance variables and therefore can encode more sophisticated extensions.

Meta-programming The fundamental work of Taha and Sheard [98] and Ancona and Moggi [9] on meta-programming and staged programming languages also appears to bear some relation to our development here. However, our focus is on isolating first-class semantic entities related to software assembly, rather than on how to express and type source (meta)-level program manipulations.

Chapter 4

Dynamic Reconfiguration

In this chapter, we focus on the issue of dynamic reconfiguration of objects. We extend our core component calculus so that the architecture of objects can be changed at run-time in a computation dependent and type safe way.

Our approach is to express dynamic reconfiguration actions by generalising the usage of configurators as *reconfiguration scripts*. Remember that configurator values are first-class values that can be created and freely combined using the mechanisms of base language. In our case, the base mechanisms are abstraction and application. Moreover, since configurators are closed values, heap independent, they can be used independently from the context of declaration.

We capture reconfiguration actions in a new language primitive, **reconfig** c[o] **in** ... , that triggers a modification described by a configuration script (a configurator yield by c) and a target object (o). This allows for reconfigurations to be programmed in ways impossible to predict in the development process.

The relation between configurators, components, and objects, with respect to the operations of composition, instantiation and reconfiguration is illustrated in Figure 4.1. The diagram shows that an object obtained from instantiating a component constructed from a configurator c and afterwards reconfigured by the configurator c', is structurally indistinguishable from an object instantiated from a component built from the composition of configurators $c; c'$. This observation accounts for the uniformity of our approach to dynamic reconfiguration.

We define a language with dynamic reconfiguration of objects, λ_ρ, based on the expressions of λ_χ^τ. We also define a type system to ensure, besides the architectural soundness of components and objects, the preservation of the soundness of reconfigured instances. We showed, in Chapter 3, that static type information is sufficient to ensure type safety of the composition of configurators, the creation of components from configurators, instantiation of components to objects, and of the access to methods in instances. However, in a setting with a strong notion of information hiding on objects such as ours, a general purpose reconfiguration mechanism is not easily verified by static type checking techniques. Remember that the structure of ob-

$$\begin{tikzcd}
c \arrow[r, "-;c'"] \arrow[d, "\textbf{compose}"'] & c;c' \arrow[d, "\textbf{compose}"] \\
C \arrow[r, dotted] \arrow[d, "\textbf{new}"'] & C' \arrow[d, "\textbf{new}"] \\
o \arrow[r, "\textbf{reconfig}\ c'[o]\ \textbf{in}\ \ldots"'] & o'
\end{tikzcd}$$

Figure 4.1: Combination of reconfiguration with composition.

jects and components is not known at compile-time. It is therefore important to explore the language design space involving combinations of static and dynamic checking. We believe to have isolated in this chapter such an interesting combination.

We rely on a run-time test that checks if each configuration script is applicable to its target instance prior to its application. This makes reconfiguration a safe operation without need for explicit error handling and resolution procedures. The test is based on the inspection of the object's structure and on a limited amount of run-time type information on both configurators and objects.

Our reconfiguration primitive follows the spirit of the early dynamic typing mechanisms in programming languages (e.g. unions in Algol68 and variant records in Pascal), later introduced in Modula3 [26] and formalised in the **typecase** construct introduced by Abadi et al. [2] to correctly deal with untyped chunks of persistent data. In these approaches to dynamic typing, the typing for the input value is assured in separate branches of the expression, the control choice over the branches depends on a run-time test on the value itself to determine the actual type (and which branch to execute next). A mechanism for the run-time inspection of types can be found, for instance, in the instanceof expression of the Java programming language.

The expression we introduce is of the form **reconfig** x = c[o] **in** ... **else** ... where expression c yields a configurator and expression o yields an object. The **in** and **else** branches specify the continuation in the two possible scenarios resulting from a run-time test between the values of configurator c and object o. Variable x is binding in both branches. This separation allows type checking both branches and thus make reconfiguration type safe.

In order to complement the examples given in Chapter 1, we illustrate the definition and application of reconfiguration actions by using a simple example where a script is used to upgrade an object. We then formally present our extended language in Section 4.2: its syntax, operational semantics, type system, and corresponding type safety result. We discuss, in Section 4.4, some details of the programming language development, and close the chapter by comparing our approach to dynamic reconfiguration with those of other authors.

4.1 Reconfiguration of a Counter Object

In this section we use a very simple example of a component implementing a counter to illustrate the features of our language and in particular emphasise the reconfiguration of objects. Let ICounter be the interface type {tick : **int** → **int**}, declaring a method tick, and consider the following definition of a component Counter:

```
let Counter = compose(
  provides p:ICounter;
  x[s:int=0,
    tick:(int → int) = fun y:int →  x.s:=x.s+y];
  plug x into p) in ...
```

As argument of the **compose** operation, we find a configuration expression, namely a sequence of operations each of which introduces a particular element of Counter: a provided port named p, a block of methods named x (implementing the method tick and a state variable s), and finally a connection between the two, using a **plug** operation. Hence, object instances of component Counter will implement a port p conforming to the interface type ICounter, and the type of Counter is the component type {}⇒{p:ICounter}, meaning that it has no required services to be instantiated, and that their instances implement, at port p, the interface ICounter. Component Counter can then be instantiated, yielding an object o, and used as in:

```
let o = new Counter in (o.p.tick(1); o.p.tick(1))
```

Component Counter may also be used as an element of other compositions, to define other components, for example, a ZeroCounter component, whose instances count all calls to tick performed with zero as argument. It is defined as follows:

```
let ZeroCounter = compose (
  provides p:ICounter;
  c[Counter:{}⇒{p:ICounter}];
  x[tick:int → int = fun y → if y=0 then c.p.tick(1) else 0];
  plug x into p) in ...
```

Here, component Counter is introduced in the structure of the configurator under the name c (in c [...]), and used in the method block (in c.p.tick(1)).

Now, suppose that a ZeroCounter object, named zc, is running on a server application, and the need arises of extending it with a new service, to reset the counter, without shutting it down: clearly, this is a situation calling for a dynamic reconfiguration facility. Consider the following (re)configuration script: (We use a while expression with the usual meaning.)

```
let AddReset = provides r:{reset:unit → unit};
               y[reset:unit → unit = while(c.p.tick(0)>0) c.p.tick(−1)];
               plug y into r) in ...
```

Configurator AddReset adds a provided port r, to expose the new reset method, implemented by the method block y. Notice that the composition operations used in the definition of AddReset refer to elements (e.g., c) which are not declared in the static context of its definition. However, the context of use is captured at the type level, with configurator AddReset being given the configurator type:

$$\{c\bullet\{p:\mathsf{ICounter}\}\}\Longrightarrow\{c\bullet\{p:\mathsf{ICounter}\},r\triangleright\{reset:unit\to unit\},y\bullet\{reset:unit\to unit\}\}$$

Remember from the definition of λ^{τ}_{χ} types, Definition 2.13, that the type of AddReset states that the configurator may be applied in every context where an element c of (object) type {p:ICounter} is present (the • resource on the left hand side). It also says that, after application, c remains available, alongside with a (new) provided port r and a (new) method block y. This type states also that AddReset may not be used to form a component, since it requires the presence of some resources in order to be applied, see Rule (Val Compose) in Figure 3.12.

Nevertheless, this configurator is fit to reconfigure objects that have an internal element named c with a compatible type. For instance, it can be used to reconfigure object zc, an instance of ZeroCounter as follows:

```
reconfig zcr = AddReset[zc] in ... zcr.r.reset() ... else ... use of zc ...
```

The expression has the effect of actually reconfiguring object zc, returning a properly typed reference zcr to the updated object that implements the reset service at a new port r. Object zc is modified in place, and therefore the behaviour of the system is changed via aliasing. Nevertheless, the known interface of the object is not changed and the execution of the remaining expressions referring zc is still type safe. The **in** branch of the **reconfig** expression is nevertheless typed with zcr having the new port r, thus its usage is type safe.

In general, a reconfiguration may not be possible, due to a mismatch between the internal architecture of the object to be reconfigured (which is not visible to the type system) and the precondition of the configurator (which is). In any case, the type system ensures that either the reconfiguration is fully applied as specified by the configurator, and the resulting object is well defined (**in** branch), or the application is not performed at all and the object is not modified (**else** branch). Moreover, because new provided ports may be added, new plug-assignments may also be necessary to satisfy newly introduced required ports. For this we use a **with** clause in the **reconfig** expression similar to that of **new**.

In summary, soundness of reconfigured instance is ensured by a direct consequence of static typing, at the level of configurator values, and of a simple and efficient test performed at reconfiguration time and based on type information recorded on objects and configurators.

4.2 λ_ρ — A Component Calculus with Dynamic Reconfiguration

In this section we present λ_ρ, a core component calculus with dynamic reconfiguration of component instances. We implement reconfiguration actions by basically applying the composition operations to the target objects. To ensure that a configurator can be applied to an object we guard the actual modification of objects by a run-time test, written $s /\!/ K$, which matches the run-time structure of the object s with a set of demanded resources K, a precondition for the application of the configuration script. Based on the run-time information that the object s is indeed fit to be reconfigured by a configurator whose type is $K \Longrightarrow K'$, for some K', we statically ensure the correctness and atomicity of the reconfiguration action.

We now define the precise syntax and semantics of λ_ρ, followed by its type system.

Syntax

Definition 4.1 (Terms). *The language λ_ρ is defined by the abstract syntax in Figure 4.2.*

We keep essentially all expressions of λ_χ^τ and introduce the new expression form $\mathsf{reconfig}$, which abstracts the application of a reconfiguration script to an object. We also add a new representation for objects, and replace the existing representation of configurator values to include manifest type information.

The reconfiguration expression $\mathsf{reconfig}\ x = e_1[e_2]\ \mathsf{with}\ \ell_i := e_i'^{\ i\in 1..n}\ \mathsf{in}\ e_3\ \mathsf{else}\ e_4$, expresses the guarded application of a configurator, yield by e_1, to an object, yield by e_2. The distinguished occurrence of x is binding, in both expression e_3 and expression e_4, denoting the target object. The reference x to the target object in e_3 is properly typed to reflect the changes to its interface that may occur in the reconfiguration action. In e_4 it is simply a reference to the unchanged object. Since the configurator e_1 may add new required ports to the instance, new values must be assigned to them by means of *plug-assignments* ($\ell_i := e_i'^{\ i\in 1..n}$).

In λ_χ^τ we encode objects using records, and for the sake of proving type safety, we distinguish record labels of required ports, provided ports, and inner elements. The role of this distinction is not semantically relevant in λ_χ^τ. However, in the present language, λ_ρ, we attach run-time type information to objects which is, in fact, essential in the semantics. Hence, we introduce a new syntactic form for object values. So, an *object* value is now represented by a triple of records of the form $(r, e, p)_\Gamma$ where the labels in r refer to its required ports, the labels in e refer to its inner elements, and the labels in p to their provided ports. Additionally Γ is a local typing environment assigning types to the internal elements of the object (Definition 2.17). Despite this distinction, and for the sake of simplicity, we sometimes refer to an object value s by the single record obtained by concatenating the three records r, e and p.

A *configurator* value, of the form $\mathsf{conf}(\tau, e)$, consists of the run-time representation of a sequence of instructions to construct or change the internal structure of an object and a configu-

e	$::=$		λ_ρ terms
	$\mid$	x	variable
	$\mid$	$\lambda x:\tau.e$	abstraction
	$\mid$	$e(e)$	application
	$\mid$	$\{\ell_i = e_i\ {}^{i\in 1..n}\}$	record
	$\mid$	$e.\ell$	selection
	$\mid$	$e.\ell := e$	assignment
	$\mid$	ℓ	port label
	$\mid$	$\mathsf{compose}\ e$	component creation
	$\mid$	$\mathsf{new}\ e\ \mathsf{with}\ \ell_j := e_j\ {}^{j\in 1..m}$	instantiation
	$\mid$	$\mathsf{reconfig}\ x = e[e]\ \mathsf{with}\ \ell_i := e_i\ {}^{i\in 1..n}\ \mathsf{in}\ e\ \mathsf{else}\ e$	reconfiguration
	$\mid$	$\mathsf{requires}\ \ell:\tau$	required port
	$\mid$	$\mathsf{provides}\ \ell:\tau$	provided port
	$\mid$	$x[e:\tau]$	component introduction
	$\mid$	$x_I[\ell_i:\tau_i = \lambda x:\tau.e_i\ {}^{i\in 1..n}]$	method block
	$\mid$	$\mathsf{plug}\ \pi:\tau\ \mathsf{into}\ \pi:\tau$	plug
	$\mid$	$e;e$	configurator composition
	$\mid$	$\imath$	location value
	$\mid$	nil	null value
	$\mid$	$(e,e,e)_\Gamma$	object
	$\mid$	$\mathsf{comp}(e)$	component
	$\mid$	$\mathsf{conf}(\tau, e)$	configurator
π	$::=$	$\ell \mid x \mid x.\ell$	port name

Figure 4.2: Abstract syntax of λ_ρ.

rator type τ that specifies the precondition on its application. Thus, configurators also embed some type information at run-time, to be used in a dynamic check, during the evaluation of reconfiguration expressions, against type information in object values.

Before introducing the operational semantics we need to define some additional notation. Let $s = (r, e, p)_\Gamma$ be an object, then we write, in uniformity with the notation on interfaces and resource sets: $r \oplus r'$ to denote the concatenation of the records r and r'; $s_\triangleleft$ to denote the record r containing the required ports in s, $s_\bullet$ to denote the available resources in record e, $s_\triangleright$ to denote the provided ports in record p, and Γ_s to denote the typing environment in s. We also write $\Gamma(s_\triangleright.\ell) = \tau$ whenever $s_\triangleright = \{\ldots, \ell = \imath, \ldots\}$ and $\Gamma(\imath) = \tau$.

$$
\begin{array}{lcll}
v & ::= & & \text{values} \\
 & \mid & \lambda x:\tau.e & \text{abstraction} \\
 & \mid & \{\ell_i = \imath_i{}^{i\in 1..n}\} & \text{record} \\
 & \mid & \imath & \text{location value} \\
 & \mid & \mathsf{nil} & \text{null value} \\
 & \mid & (r,r,r)_\Gamma & \text{object} \\
 & \mid & \mathsf{conf}(\tau,c) & \text{configurator} \\
 & \mid & \mathsf{comp}(c) & \text{component} \\
 & & & \\
r & ::= & \{\ell_i = \imath_i{}^{i\in 1..n}\} & \text{record} \\
 & & & \\
c & ::= & & \text{ground composition operations} \\
 & \mid & \mathsf{requires}\ \ell:\tau & \text{required port} \\
 & \mid & \mathsf{provides}\ \ell:\tau & \text{provided port} \\
 & \mid & x[v:\tau] & \text{inner component} \\
 & \mid & x_I[\ell_i:\tau_i = \lambda x_i.e_i{}^{i\in 1..n}] & \text{method block} \\
 & \mid & \mathsf{plug}\ \pi:\tau\ \mathsf{into}\ \pi:\tau & \text{plug} \\
 & \mid & c;c & \text{composition} \\
 & & & \\
\pi & ::= & \ell \mid x \mid x.\ell & \text{port name}
\end{array}
$$

Figure 4.3: Abstract syntax of λ_ρ values.

Definition 4.2 (Notation).

- *Given the records* $\{\ell_i = \imath_i{}^{i\in 1..n}, \ell'_i = \imath''_i{}^{i\in 1..k}\}$ *and* $\{\ell'_j = \imath_j{}^{j\in 1..m}\}$ *with* $k \leq m$ *we have* $\{\ell_i = \imath_i{}^{i\in 1..n}, \ell'_i = \imath''_i{}^{i\in 1..k}\} \oplus \{\ell'_j = \imath_j{}^{j\in 1..m}\} \triangleq \{\ell_i = \imath_i{}^{i\in 1..n}, \ell'_j = \imath_j{}^{j\in 1..m}\}$.
- *If* $s = (r,e,p)_\Gamma$ *then*
  - $s_\triangleleft \triangleq r$.
  - $s_\triangleright \triangleq p$.
  - $s_\bullet \triangleq r \oplus e$.
  - $\Gamma_s \triangleq \Gamma$
  - $\Gamma(s_\triangleright.\ell) = \tau$ *if* $s_\triangleright = \{\ldots, \ell = \imath, \ldots\}$ *and* $\Gamma(\imath) = \tau$.

We now define the expressions which may result from evaluating an expression of λ_ρ.

Definition 4.3 (Values). *The set of values* $\mathcal{U}_\rho \subseteq \lambda_\rho$ *is defined by the abstract syntax in Figure 4.3.*

The values of λ_ρ differ from the values of λ^τ_χ in the representation for objects and configurators. Notice that c refers to the same set of ground composition operations as it does in λ^τ_χ.

(Match Provides)

$$\frac{\Gamma_s(s_{\triangleright}.\ell) = \tau \quad s /\!/ K}{s /\!/ \ell \triangleright \tau, K}$$

(Match Requires)

$$\frac{\Gamma_s(s_{\triangleleft}.\ell) = \tau \quad s /\!/ K}{s /\!/ \ell \triangleleft \tau, K}$$

(Match Element)

$$\frac{\Gamma_s((s_{\triangleleft} \oplus s_{\bullet}).\ell) = \tau \quad s /\!/ K}{s /\!/ \ell \bullet \tau, K}$$

(Match Element Port)

$$\frac{S(s_{\bullet}.x) = s' \quad \Gamma_{s'}({s'}_{\triangleright}.\ell) = \tau \quad s /\!/ K}{s /\!/ x.\ell \bullet \tau, K}$$

(Match Unsatisfied)

$$\frac{\Gamma_s(s_{\triangleright}.\ell) = \tau \quad s /\!/ K}{s /\!/ \ell \circ \tau, K}$$

(Match Unsatisfied Port)

$$\frac{S(s_{\bullet}.x) = s' \quad \Gamma_{s'}({s'}_{\triangleleft}.\ell) = \tau \quad s /\!/ K}{s /\!/ x.\ell \circ \tau, K}$$

Figure 4.4: Matching rules for λ_ρ.

Operational Semantics

We now define the operational semantics of λ_ρ with the elements necessary for the evaluation of reconfiguration expressions. As explained above, type safety of dynamic reconfiguration relies on a run-time check which is here formalised by a matching test relation.

We thus define the matching test, which checks whether a configurator is compatible with the structure of a given instance, and therefore applicable. We use the term "applicable" informally to mean that the application of the configurator does not disrupt the structure of the target instance. In formal terms, we specify that a configurator with precondition type K is applicable to s if the matching test $s /\!/ K$, defined next, holds. Our type safety result will show that this informal and formal definitions are two sides of the same coin.

Definition 4.4 (Matching)**.** *Given an object s defined with relation to a heap S we say that s matches a set of resources K, if $s /\!/ K$ is derivable by the rules in Figure 4.4.*

Intuitively, an instance $s = (r, e, p)_\Gamma$ matches a set of resources K if each one of the resources in K can be found, with compatible types, in one particular record: r, e, or p. Notice in particular that:

- The presence of a compatible resource denoting a provided port $\ell \triangleright \tau$ is checked in record $s_{\triangleright}$, Rule (Match Provides);

- The presence of a compatible resource denoting a required port $\ell \triangleleft \tau$ is checked in record $s_{\triangleleft}$, Rule (Match Requires);

- An available element denoted by a label ℓ is looked for in either the required ports or the internal elements (denoting a scripting block), Rule (Match Element);

(Eval Requires)$^\rho$

$$\mathsf{requires}\ \ell : \tau; S \downarrow \mathsf{conf}(\emptyset \Longrightarrow \{\ell \bullet \tau, \ell \lhd \tau\}, \mathsf{requires}\ \ell : \tau); S$$

(Eval Provides)$^\rho$

$$\mathsf{provides}\ \ell : \tau; S \downarrow \mathsf{conf}(\emptyset \Longrightarrow \{\ell \circ \tau, \ell \rhd \tau\}, \mathsf{provides}\ \ell : \tau); S$$

(Eval Plug)$^\rho$

$$\mathsf{plug}\ \pi_1 : \tau\ \mathsf{into}\ \pi_2 : \tau; S \downarrow \mathsf{conf}(\{\pi_2 \circ \tau, \pi_1 \bullet \tau\} \Longrightarrow \{\pi_1 \bullet \tau\}, \mathsf{plug}\ \pi_1 : \tau\ \mathsf{into}\ \pi_2 : \tau); S$$

(Eval Sequence)$^\rho$

$$\frac{e_1; S \downarrow \mathsf{conf}(K \Longrightarrow K', K_c\ ; c_1); S' \quad e_2; S \downarrow \mathsf{conf}(K_c, K'' \Longrightarrow K'''\ ; c_2); S'}{(e_1; e_2); S \downarrow \mathsf{conf}(K, K'' \Longrightarrow K', K'''\ ; (c_1; c_2)); S'}$$

(Eval Uses)$^\rho$ $(\tau = \{\ell^r_i : \tau_i\ ^{i \in 1..k}\} \Rightarrow \{\ell^p_j : \sigma_j\ ^{j \in 1..m}\})$

$$\frac{e; S \downarrow v; S'}{x[e : \tau]; S \downarrow \mathsf{conf}(\emptyset \Longrightarrow \{x \bullet \sigma, x.\ell^r_i \circ \tau_i\ ^{i \in 1..n}, x.\ell^p_j \bullet \sigma_j\ ^{j \in 1..m}\}, x[v : \tau]); S'}$$

(Eval Method Block)$^\rho$ $(I = \{\ell'_i : \tau'_i\ ^{i \in 1..n}\},\ K = \{\ell'_i \bullet \tau'_i\ ^{i \in 1..n}\})$

$$x_I[\ell_i : \tau_i = v_i\ ^{i \in 1..n}]; S \downarrow \mathsf{conf}(K \Longrightarrow K, \{x \bullet \{\ell_i : \tau_i\ ^{i \in 1..n}\}\}\ ;\ x_I[\ell_i : \tau_i = v_i\ ^{i \in 1..n}]); S$$

Figure 4.5: Evaluation rules for λ_ρ.

- Available resources with compound names ($x.\ell \bullet \tau$) can only be related to provided ports of inner elements, Rule (Match Element Port)
- Simple port names tagged as unsatisfied ($\ell \circ \tau$) can only refer to provided ports (from the inside perspective of a composition), Rule (Match Unsatisfied);
- and unsatisfied compound port names ($x.\ell \circ \tau$) can only be required ports of inner components, Rule (Match Unsatisfied Port).

Notice also that the heap referred in the rules in Figure 4.4 is derived from the evaluation context. We avoid tagging every rule with the heap, to make notation lighter without loss of clarity. We prove later that the run-time success of this test implies the notion of architectural conformance, between an object and a resource set, which is used in Section 3.2 to prove subject reduction of λ^τ_χ (Definition 3.23).

Having defined this essential part of the semantics, the evaluation of λ_ρ expressions is now defined in the following way:

Definition 4.5 (Evaluation). *Let $e \in \lambda_\rho$ and a heap S such that $(e; S)$ is a valid configuration. The evaluation relation of an expression e to a value v, written $e; S \downarrow v; S'$ is defined inductively by the rules in Figures 2.8, 4.4, 4.5, 4.6, and 4.7.*

(Eval Compose)$^\rho$

$$\frac{e; S \downarrow \mathsf{conf}(\tau, c); S'}{\mathsf{compose}\ e; S \downarrow \mathsf{comp}(c); S'}$$

(Eval New) $\quad (s = (r, e, p), \quad r = \{\ell_i = \iota_i\ ^{i\in 1..n}\}, \quad \iota = \mathrm{new}(S))$

$$\frac{e; S \downarrow \mathsf{comp}(c); S_0 \quad \mathbf{0}; c; S_n \Downarrow s; S_{n+1} \quad e_i; S_{i-1} \downarrow v_i; S_i\ \forall_{i\in 1..n}}{\mathsf{new}\ e\ \mathsf{with}\ \ell_i := e_i\ ^{i\in 1..n}; S \downarrow \iota; S_{n+1}[\iota \mapsto s][\iota_i \mapsto v_i\ ^{i\in 1..n}]}$$

(Eval Reconfig) $\qquad (s'_{\triangleleft} = s_{\triangleleft} \oplus \{\ell_i = \iota_i\ ^{i\in 1..n}\})$

$$\frac{\begin{array}{l} e_1; S \downarrow \mathsf{conf}(K \Longrightarrow K', c); S' \\ e_2; S' \downarrow \iota; S_0 \quad s = S_0(\iota) \quad s /\!/ K \\ s; c; S_n \Downarrow s'; S_{n+1} \\ f_i; S_{i-1} \downarrow v_i; S_i \quad \forall_{i\in 1..n} \\ e_3[x \leftarrow \iota']; S_{n+1}[\iota' \mapsto s'][\iota_i \mapsto v_i\ ^{i\in 1..n}] \downarrow v; S'''' \end{array}}{\mathsf{reconfig}\ x = e_1[e_2]\ \mathsf{with}\ \ell_i := f_i\ ^{i\in 1..n}\ \ \mathsf{in}\ e_3\ \mathsf{else}\ e_4; S \downarrow v; S'''}$$

(Eval Reconfig Else)

$$\frac{\begin{array}{l} e_1; S \downarrow \mathsf{conf}(K \Longrightarrow K', c); S' \\ e_2; S' \downarrow \iota; S'' \quad s = S''(\iota) \quad \neg s /\!/ K \\ e_4[x \leftarrow \iota]; S'' \downarrow v; S''' \end{array}}{\mathsf{reconfig}\ x = e_1[e_2]\ \mathsf{with}\ \ell_i := f_i\ ^{i\in 1..n}\ \ \mathsf{in}\ e_3\ \mathsf{else}\ e_4; S \downarrow v; S'''}$$

Figure 4.6: Evaluation rules for λ_ρ (part 2).

We define the operational semantics of λ_ρ given the evaluation rules for λ_R expressions and the rule for new expressions in λ_χ^τ. We modify the evaluation of the compose e expression, to accommodate the syntactic changes made to configurator values, and change the evaluation of composition operations to compute the necessary run-time type information. We also change the application of configurators to composition contexts to accommodate these changes. We finally define the evaluation of reconfig $r[o]$ in ... else ... expressions.

Rule (Eval Compose)$^\rho$ constructs component values from configurators by "forgetting" the run-time type information they enclose and copying their composition operation.

The evaluation of composition expressions is defined in such a way that it incrementally builds configurators with a manifest type expression and a ground composition operation (Figure 4.5). The manifest type information that we represent here by a type expression, miming the typing process already defined in λ_χ^τ (Definition 3.18), is confined to the values themselves and can be seen in analogy with signature or hash mechanisms usually incorporated in main-stream middleware frameworks that dynamically manipulate data and code. e.g. Stream Unique Identifiers (SUID) in Java (i.e. an hash of the class name, interface class names, methods, and fields), globally unique identifiers (GUID) in COM and .NET.

Notice the compositional construction of such information in Rule (Eval Sequence): a two part composition gets the type information obtained from blindly composing the two corre-

$$\text{(App Requires)}^\rho \qquad (\imath = \text{new}(S))$$
$$(r, e, p)_\Gamma; (\mathsf{requires}\ \ell : \tau); S \Downarrow (r \oplus \{\ell = \imath\}, e, p)_{\Gamma, \imath:\tau}; S[\imath \mapsto \mathsf{nil}]$$

$$\text{(App Provides)}^\rho \qquad (\imath = \text{new}(S))$$
$$(r, e, p)_\Gamma; (\mathsf{provides}\ \ell : \tau); S \Downarrow (r, e, p \oplus \{\ell = \imath\})_{\Gamma, \imath:\tau}; S[\imath \mapsto \mathsf{nil}]$$

$$\text{(App Uses)}^\rho \quad \frac{\mathsf{new}\ v; S \downarrow \imath; S'}{(r, e, p)_\Gamma; x[v : \tau \Rightarrow \sigma]; S \Downarrow (r, e \oplus \{x = \imath\}, p)_{\Gamma, \imath:\sigma}; S'} \qquad \text{(App Sequence)}^\rho \quad \frac{s; c_1; S \Downarrow s'; S' \quad s'; c_2; S' \Downarrow s''; S''}{s; (c_1; c_2); S \Downarrow s''; S''}$$

$$\text{(App Method Block)}^\rho \quad (\imath,\ \imath_i\ ^{i \in 1..n} = \text{new}(S),\ \ v_i' = v_i[(r, e, p)_\Gamma][x \leftarrow \imath],\ \ \Gamma' = \Gamma, \imath : \{\!|\ell_i : \tau_i\ ^{i \in 1..n}|\!\})$$
$$(r, e, p)_\Gamma; x_I[\ell_i{:}\tau_i = v_i\ ^{i \in 1..n}]; S \Downarrow (r, e \oplus \{x = \imath\}, p)_{\Gamma'}; S[\imath \mapsto \{\ell_i \mapsto \imath_i\ ^{i \in 1..n}\}][\imath_i \mapsto v_i'\ ^{i \in 1..n}]$$

$$\text{(App Plug)}^\rho$$
$$s; \mathsf{plug}\ \pi_1{:}\tau_1\ \mathsf{into}\ \pi_2{:}\tau_2; S \Downarrow s; S[\text{select}_S(s, \pi_2) \mapsto \text{select}_S(s, \pi_1)]$$

Figure 4.7: Application rules for composition operations in λ_ρ.

sponding run-time type information pieces. No code inspection occurs in the composition process. This can be seen as a low-level composition of the two manifest type information bits.

Notice that this composition mechanism trusts that the implementation of the base elements indeed corresponds to their type description. Testing the actual code against these manifest information is dependent on the actual run-time representation of composition operations. In this case, where we store the ground composition operations inside configurator values, the certification process is defined simply by replaying the typing process at load-time. In other cases where the composition operation may be "compiled" in some way, an approach based on works on proof carrying code [75] can be followed. The certification of configurator values can orthogonally added to the loading mechanisms of such a system, provided that there is a composition mechanism for the proofs (the types) as well as the code itself.

As explained before, a reconfiguration operation depends on a run-time test to check that a configurator is in fact compatible with the structure of a given instance. The evaluation of a reconfig expression is thus defined by two rules to cover the two possible outcomes of a matching test, (Eval Reconfig) and (Eval Reconfig Else). The former is applicable whenever the test $s /\!/ K$ succeeds, where s is the target instance and K the precondition present in the type of the configurator. The composition operation c, taken from the configurator yield by e_1 is then applied to s, the instance obtained from e_2. The final result comes from evaluating e_3. Rule (Eval Reconfig Else) is applicable otherwise, it follows by evaluating the **else** branch. Notice that only the required resources in the configurator's run-time type information (the precondition) are used to test the instance. Nevertheless, the added resources (the postcondition of

(Val Reconfig) $\left(K'_\circ = \emptyset,\ K'_\triangleright \# I,\ K'_\triangleleft = \{\ell_i : \sigma_i\ ^{i\in 1..n}\}\right)$

$$\frac{\begin{array}{c}\Delta \vdash e_1 : K \Longrightarrow K' \quad \Delta \vdash e_2 : I \quad \Delta \vdash e'_i : \sigma_i\ \ \forall_{i\in 1..n} \\ \Delta, x : I \oplus K'_\triangleright \vdash e_3 : \delta \quad \Delta, x : I \vdash e_4 : \delta\end{array}}{\Delta \vdash \mathsf{reconfig}\ x = e_1[e_2]\ \mathsf{with}\ \ell_i := e'_i\ ^{i\in 1..n}\ \mathsf{in}\ e_3\ \mathsf{else}\ e_4 : \delta}$$

(Val Object)

$$\frac{s_\triangleright = \{\ell_i = l_i\ ^{i\in 1..n}\} \quad \Gamma(l_i) = \tau_i \quad \Delta(l_i) = \tau_i\ \ \forall_{i\in 1..n}}{\Delta \vdash s : \{\ell_i : \tau_i\ ^{i\in 1..n}\}}$$

(Val Configurator Value)$^\rho$

$$\frac{\Delta \vdash c : K \Longrightarrow K'}{\Delta \vdash \mathsf{conf}(K \Longrightarrow K', c) : K \Longrightarrow K'}$$

Figure 4.8: Typing rules for λ_ρ.

the configurator) are important in the process of building the type information. Notice, in Rule (Eval Sequence) in Figure 4.5, that the set of resources K_c, which is a provided resource set of the first configurator is essential to compute the resulting configurator type.

Moreover, the application of configurators to composition contexts is also modified with relation to λ^τ_χ in order to build the necessary run-time type information in the resulting object. Notice that this information is also confined to the instance, and is only used in the matching test. Again, this results from a context free, mechanical process that gathers type annotations from the composition expressions and accumulates them in the typing environment in the object value.

We next define a type system for λ_ρ that, despite the dynamic character of reconfiguration, ensures the soundness of components and objects. This soundness properties also hold in the case of reconfigured objects.

Type System

We uniformly extend the typing relation of λ^τ_χ with the Rules in Figure 4.8. Rule (Val Reconfig) to type reconfig expressions and Rules (Val Object) and (Val Configurator Value)$^\rho$ to type object and configurator values.

Definition 4.6 (Typing relation)**.** *The judgement $\Delta \vdash e : \tau$ is valid if it is derivable by the rules in Figures 2.11, 3.12, 3.13 and 4.8.*

Despite the dependence of reconfiguration actions on a run-time check, some basic conformance between the configurator type ($K \Longrightarrow K'$) and the type of the target object (τ) is tested statically in Rule (Val Reconfig). We use K' to ensure that the continuations of reconfiguration actions are well-typed, and, in particular, ensure, at the level of the reconfigured instance, that:

- no dependencies are left open after the application ($K'_\circ = \emptyset$),
- the configurator does not override the object's provided ports ($K'_\triangleright \# I$), and

- all new required ports must be satisfied by plug-assignments, ($K'_\triangleleft = \{\ell_i : \sigma_i {}^{i\in 1..n}\}$).

Rule (Val Object) and Rule (Val Configurator Value)$^\rho$ assert that the corresponding values are well-formed. We define object and configurator values to be well-formed with relation to a typing environment if they carry the expected type information. In particular, we see here that well-typed programs always produce configurators whose composition operation conforms with the run-time type information.

We illustrate the typing and evaluation relations in a simple example of a reconfiguration:

Example 4.7. Consider the types τ, $I = \{f_1 : \tau \to \tau\}$, and $I' = \{f_2 : \tau \to \tau \times \tau\}$, and the values v_1 and v_2 with type τ, in the expressions to follow. We also use standard forms for product types and value constructors. Consider the following expression:

$$(1)\quad \begin{array}{l} \mathsf{let}\ c = \mathsf{compose}\ (\mathsf{provides}\ p : I;\ m_1[f_1 : (\tau \to \tau) = \lambda x : \tau.x];\ \mathsf{plug}\ m\ \mathsf{into}\ p)\ \mathsf{in} \\ \mathsf{let}\ o = \mathsf{new}\ c\ \mathsf{in} \\ o.p.f_1(v_1). \end{array}$$

Here, c has type $\{\} \Rightarrow \{p : I\}$ and o has type $\{p : I\}$. The expression $o.p.f_1(v)$ representing here the regular use of object o has type τ.

Now, suppose the implementation of m_1 is faulty and we want to replace it in o by a method block that replaces m_1 and, at the same time, implements the functionality for a new port. We may apply the following reconfiguration script to the object o:

$$(2)\quad \begin{array}{l} \mathsf{let}\ r = (m_2[f_1 : (\tau \to \tau) = \lambda x : \tau.v_2, \\ \qquad\qquad f_2 : (\tau \to \tau \times \tau) = \lambda x : \tau.(x, x)]; \\ \qquad\quad \mathsf{plug}\ m_2\ \mathsf{into}\ p; \\ \qquad\quad \mathsf{provides}\ q : I'; \\ \qquad\quad \mathsf{plug}\ m_2\ \mathsf{into}\ q)\ \mathsf{in} \\ \mathsf{reconfig}\ x = r[o]\ \mathsf{in}\ x.q.f_2(v_1)\ \mathsf{else}\ (v_1, v_2). \end{array}$$

In this case the type of the reconfiguration script r is $\{p \circ I, m_1 \bullet I\} \Longrightarrow \{m_2 \bullet I, q \triangleright I'\}$. This expression is successfully typed with relation to the typing environment $\Delta = c : \{\} \Rightarrow \{p : I\}, o : \{p : I\}, r : \{p \circ I, m_1 \bullet I\} \Longrightarrow \{m_2 \bullet I, q \triangleright I'\}$ by the following derivation:

(Val Var)	$\Delta \vdash r : \{p \circ I, m_1 \bullet I\} \Longrightarrow \{m_2 \bullet I, q \triangleright I'\}$
(Val Var)	$\Delta \vdash o : \{p : I\}$
(Val Var)	$\Delta, x : \{p : I, q : I'\} \vdash x : \{p : I, q : I'\}$
(Val Select)	$\Delta, x : \{p : I, q : I'\} \vdash x.q : I'$
(Val Select)	$\Delta, x : \{p : I, q : I'\} \vdash x.q.f_2 : \tau \rightarrow \tau \times \tau$
...	$\Delta, x : \{p : I, q : I'\} \vdash v_1 : \tau$
(Val Application)	$\Delta, x : \{p : I, q : I'\} \vdash x.q.f_2(v_1) : \tau \times \tau$
...	$\Delta, x : \{p : I\} \vdash (v_1, v_2) : \tau \times \tau$
	$(\{m_2 \bullet I, q \triangleright I'\}_{\circ} = \emptyset,\ \ \{m_2 \bullet I, q \triangleright I'\}_{\triangleleft} = \emptyset,\ \ \{m_2 \bullet I, q \triangleright I'\}_{\triangleright} \# \{p : I\} = \emptyset)$
(Val Reconfig)	$\Delta \vdash \textsf{reconfig}\ x = r[o]\ \textsf{in}\ x.q.f_2(v_1)\ \textsf{else}\ (v_1, v_2) : \tau \times \tau$

In fact, if we simulate the evaluation of expression (1) we obtain that o denotes a location l_2 in the heap:

$$
\begin{aligned}
S = \{\, & l_0 \mapsto \{f_1 = \lambda x : \tau.x\}, \\
& l_1 \mapsto l_0 \\
& l_2 \mapsto (\{\}, \{m_1 = l_0\}, \{p = l_1\})_{l_0:I, l_1:I} \}.
\end{aligned}
$$

In this case, the test $S(l_2) /\!/ \{p \circ I\}$ succeeds, and when r is in fact applied to o, the resulting location $x = l_5$ denotes an instance in the heap

$$
\begin{aligned}
S = \{\, & l_0 \mapsto \{f_1 = \lambda x : \tau.x\}, \\
& l_1 \mapsto l_3 \\
& l_2 \mapsto (\{\}, \{m_1 = l_0\}, \{p = l_1\})_{l_0:I, l_1:I} \\
& l_3 \mapsto \{f_1 = \lambda x : \tau.v_2,\ \ f_2 = \lambda x : \tau.(x, x)\}, \\
& l_4 \mapsto l_3, \\
& l_5 \mapsto (\{\}, \{m_1 = l_0, m_2 = l_3\}, \{p = l_1, q = l_4\})_{l_1:I, l_2:I', l_3:I, l_4:I'} \}.
\end{aligned}
$$

In (2) $x.q$ is mapped to the record referred by l_3 and the call $x.q.f_2(v_1)$ succeeds in accessing f_2 in method block m_2. Notice that the location l_0 is no longer referred by any provided port, thus being inaccessible unless references exist outside the instance or some other reconfiguration operation reconnects it to a provided port.

Having illustrated the interaction between typing and evaluation we now formally connect the two relations and prove type safety in a subject reduction theorem.

4.2.1 Type Safety

In this section we prove that, besides the construction properties ensured in λ_{χ}^{τ} by Theorem 3.30, the typing relation of λ_{ρ}, also ensures the soundness of newly reconfigured instances

(Wrong Reconfig)

$$\frac{e_1; S \downarrow v; S' \quad v \neq \mathsf{conf}(\tau, c)}{\mathsf{reconfig}\ x = e_1[e_2]\ \mathsf{with}\ r_i := e_i'^{\,i \in 1..n}\ \mathsf{then}\ e_3\ \mathsf{else}\ e_4; S \downarrow \mathsf{wrong}; S'''}$$

(Wrong Reconfig 2)

$$\frac{e_1; S \downarrow \mathsf{conf}(\tau, c); S' \quad e_2; S' \downarrow v; S'' \quad v \notin \mathsf{Loc}}{\mathsf{reconfig}\ x = e_1[e_2]\ \mathsf{with}\ r_i := e_i'^{\,i \in 1..n}\ \mathsf{then}\ e_3\ \mathsf{else}\ e_4; S \downarrow \mathsf{wrong}; S'''}$$

(Wrong Reconfig 3)

$$\frac{e_1; S \downarrow \mathsf{conf}(\tau, c); S' \quad e_2; S' \downarrow \iota; S'' \quad S(\iota) \neq (r, e, p)_\Gamma}{\mathsf{reconfig}\ x = e_1[e_2]\ \mathsf{with}\ r_i := e_i'^{\,i \in 1..n}\ \mathsf{then}\ e_3\ \mathsf{else}\ e_4; S \downarrow \mathsf{wrong}; S'''}$$

Figure 4.9: Error trapping rules for λ_ρ.

with relation to its operational semantics. We follow the same technique used in Sections 2.2.1 and 3.2.1 to prove subject reduction. We enunciate a theorem that asserts the preservation of types throughout the evaluation and the absence of run-time errors on well-typed expressions that may be caused by ill-formed architectures. The soundness of reconfigured instances follows as logic consequence of the subject reduction.

Again, for proving type safety in λ_ρ we consider the extended operational semantics with a distinguished value, wrong, to which expressions evaluate whenever a run-time error occurs.

The extended operational semantics is defined as follows, based on the rules in Figure 4.9.

Definition 4.8 (Extended Evaluation). *Let $e \in \lambda_\rho$ and a heap S such that $(e; S)$ is a valid configuration. The evaluation relation of an expression e to a value v, written $e; S \downarrow v; S'$ is defined inductively by the rules in Figures 2.8, 2.12, 4.4, 4.5, 4.6, 4.7, 3.14, and 4.9.*

Recall our definition of architecture conformance from λ_χ^τ, Definition 3.23. It asserts that an object is partially described by a particular resource set. Remember that this is central in the invariant of the induction in the subject reduction proof, stating that, at all times, the partially built instances conform with the type information statically assigned to the construction operations. We now correlate the architectural conformance property it with the matching test used in the operational semantics, Definition 4.4. We then use this correlation as the starting point for the application of a configurator to a running instance in Rule (Eval Reconfig). This property is expressed in the following lemma:

Lemma 4.9. *For all instances s and resource sets K, if $s \mathbin{/\!/} K$ then s conforms with K.*

Proof. The proof is done by induction on the height of the derivations of $s \mathbin{/\!/} K$ with $s = (r, e, p)_\Gamma$ and by case analysis on the last rule used. (for the complete proof see appendix A on page 185). □

As before, we state the usual properties of weakening and preservation of types under substitution of variables are essential to prove type safety.

Lemma 4.10 (Weakening). *For all typing environments Δ, Δ', all expressions $e \in \lambda_\rho$, and $\tau \in \mathcal{T}_\chi$: If $\Delta, \Delta' \vdash e : \tau$ and $x \notin Dom(\Delta')$ then $\Delta, x : \tau', \Delta' \vdash e : \tau$.*

Proof. By induction on the height of the derivations and in the cases of the last rule used. □

Lemma 4.11 (Substitution). *For all typing environments Δ, Δ', all expressions $e \in \lambda_\rho$, and $\tau \in \mathcal{T}_\chi$: If $\Delta, x : \tau, \Delta' \vdash e : \tau'$ and $\Delta \vdash v : \tau$ then $\Delta, \Delta' \vdash e[x \leftarrow v] : \tau'$.*

Proof. By induction on the height of the derivations and in the cases of the last rule used. □

We are now ready to enunciate the general lemma that asserts subject reduction considering simultaneously the evaluation of expressions and the application of composition operations.

Lemma 4.12 (Subject Reduction).

1. *Let $(e; S)$ be a valid configuration in $\lambda_\rho \backslash \{\mathsf{nil}\}$ such that $\mathsf{nil}(S) = \emptyset$ and let Γ be a typing environment typing S such that e and S are record-based with relation to Γ.:*

   If $\Gamma \vdash e : \tau$ and $(e; S \downarrow v; S')$ then

   a) *there is a Γ' that extends Γ and types S',*
   b) *$\Gamma' \vdash v : \tau$,*
   c) *v is either an abstraction, a component, a configurator, or a location that is either `undefined` or `refers-to` a record,*
   d) *v and S' are record-based with relation to Γ, and*
   e) *$\mathsf{nil}(S') = \emptyset$.*

2. *Let c be an expression such that $\Gamma \vdash c : K \Longrightarrow K'$, let S be a heap such that, for some set $X \in Dom(S)$, $\mathsf{nil}(S) \subseteq \{\mathrm{select}_S(s, \pi) \mid (\pi : \tau) \in K_\triangleleft \cup K_\circ\} \uplus X$ and Γ types S.*

   Let s be a partially linked object s such that it conforms with K and its partially linked object type is $[\![R \oplus K_\triangleleft \Rightarrow P \oplus K_\triangleright]\!]$ and s and S are record-based with relation to Γ:

   If $s; c; S \Downarrow s'; S'$ then

   a) *there is Γ' typing S' and extending Γ,*
   b) *s' is a partially linked object that extends s and conforms with K'. Its partially linked object type is $[\![R \oplus K'_\triangleleft \Rightarrow P \oplus K'_\triangleright]\!]$, and*
   c) *s' and S' are record-based with relation to Γ', and*
   d) *$\mathsf{nil}(S') \subseteq \{\mathrm{select}_{S'}(s, \pi) \mid (\pi : \tau) \in K'_\triangleleft \cup K'_\circ\} \uplus X$.*

Proof Sketch. This lemma is proven by induction on the height of the evaluation derivations and by case analysis on the last rule used. The cases of interest in this proof are (Eval Reconfig) and (Eval Reconfig Else), all other cases follow by induction on the size of the evaluation derivations, just like in the proof of Lemma 3.29 for λ^τ_χ. They are only different in handling the manipulation of the manifest type information in objects and configurators.

***Case:* (Eval Reconfig)**

If the last rule used is (Eval Reconfig) then we must have

a) $\mathsf{reconfig}\ x = e_1[e_2]\ \mathsf{with}\ \ell_i := f_i\ ^{i\in 1..n}\ \ \mathsf{in}\ e_3\ \mathsf{else}\ e_4; S \downarrow v; S'''$

and

b) $\Gamma \vdash \mathsf{reconfig}\ x = e_1[e_2]\ \mathsf{with}\ \ell_i := e'_i\ ^{i\in 1..n}\ \mathsf{in}\ e_3\ \mathsf{else}\ e_4 : \delta$.

From b), by Rule (Val Reconfig), we have that $\Gamma \vdash e_1 : K \implies K'$ and from a), by Rule (Eval Reconfig), we have $e_1; S \downarrow \mathsf{conf}(\tau', c); S'$. By induction hypothesis, we conclude that there is a Γ' that extends Γ and types S' such that $\Gamma' \vdash \mathsf{conf}(\tau', c) : K \implies K'$, which, by Rule (Val Configurator Value), leads to $\tau' = K \implies K'$ and $\Gamma' \vdash c : \tau'$. From b), by Rule (Val Reconfig), we also have that $\Gamma \vdash e_2 : \tau$. By Lemma 4.10 (weakening) we obtain $\Gamma' \vdash e_2 : \tau$, and $e_2; S' \downarrow \iota; S_0$ with $S_0(\iota) = s$. By induction hypothesis we conclude that there is a Γ'' which extends Γ' and types S_0 such that $\Gamma'' \vdash \iota : \tau$.

Again, from a), by Rule (Eval Reconfig), we now know that $s \mathbin{//} K$ and therefore, by Lemma 4.9, we know that s conforms with K and should have a partial linked object type $[\![\sigma \Rightarrow \tau]\!]$ for some object type σ.

The evaluation follows on the plug assignment expressions. By iterating Lemma 4.10 (weakening) and the induction hypothesis for all the expressions e_i, typed by $\Gamma_i \vdash e_i : \sigma_i$, we obtain that for each i there is a Γ_i that extends Γ_{i-1}, with $\Gamma_0 = \Gamma''$ typing each S_i such that $\Gamma_i \vdash v_i : \sigma_i$.

Then, we have that $s; c; S_n \Downarrow s'; S''$. By induction hypothesis on the second part of the lemma (the application of a configurator) we have that there is a Γ''' which extends Γ'' and that the resulting instance, s', conforms with K' and has the partial type $[\![K'_{\triangleleft} \oplus \sigma \Rightarrow K'_{\triangleright} \oplus \tau]\!]$ and its object type is $K'_{\triangleright} \oplus \tau$.

Let $\Gamma'''' = \Gamma''', \iota' : (\tau \oplus K'_{\triangleright})$. Again, from b), by Rule (Val Reconfig) and Lemma 4.10 (weakening), we obtain $\Gamma'''', x : (\tau \oplus K'_{\triangleright}) \vdash e_3 : \delta$ and by Lemma 4.11 (substitution) with the side condition $\Gamma'''' \vdash \iota' : (\tau \oplus K'_{\triangleright})$. We conclude c) $\Gamma'''' \vdash e_3[x \leftarrow \iota'] : \delta$. So, from a), by Rule (Eval Reconfig), we have $e_3[x \leftarrow \iota']; S_{n+1}[\iota' \mapsto s'][\iota_i \mapsto v_i\ ^{i\in 1..n}] \downarrow v; S''''$, and with c), by induction hypothesis, we obtain the final result that there is a Γ'''' such that $\Gamma'''' \vdash v : \delta$ with Γ'''' extending Γ''' types S''''.

Finally, the occurrence of nil values in the heap follows the reasoning used in the instantiation process. In the first case, all nil values introduced by the new required ports are replaced by non-nil values, so we prove that $\mathsf{nil}(S'''') = \emptyset$. The new values introduced in the heap are all record-based (by induction hypothesis) and so it is the result of evaluating e_3.

Case: **(Eval Reconfig Else)**

The second evaluation possibility for a reconfiguration follows without applying the configurator to the instance and therefore the typing of the resulting value v results by induction hypothesis on the typing derivation of e_4.

To complete the proof of type safety we must analyse the cases where the result of evaluating an expression may be a wrong value, and prove that it does never occur when evaluating a well-typed expressions. For this we analyse the rules in Figures 2.12, 3.14, and 4.9 and conclude that they are not applicable. (for the complete proof see appendix A on page 186). □

Finally, the main result of this chapter follows as a corollary of lemma 4.12. The subject reduction theorem yields type safety for the language λ_ρ as follows:

Theorem 4.13 (Subject Reduction)**.** *Let $(e; S)$ be a valid configuration in $\lambda^{\tau}_{\chi}\backslash\{\mathsf{nil}\}$ such that $\mathsf{nil}(S) = \emptyset$ and let Γ be a typing environment typing S such that e and S are record-based with relation to Γ. If $\Gamma \vdash e : \tau$ and $e; S \downarrow v; S'$ then*

a) there is a Γ' that extends Γ and types S',

b) $\Gamma' \vdash v : \tau'$,

c) v is either an abstraction, a component, a configurator, or a location that is either `undefined` or `refers-to` a record,

d) If v is a component or configurator value then $FL(v) = \emptyset$, and

e) $\mathsf{nil}(S') = \emptyset$.

Proof. This theorem results directly from the first case of Lemma 4.12 and Lemma 3.25. □

This theorem ensures that after a reconfiguration, the target instance has a sound architecture, i.e. that all provided ports of instances actually lead to a record containing the expected set of methods, and also that all references to external functionality are indeed fulfilled and so, that all expressions declared inside method blocks do not fail due to ill-formed architectures.

From the proof of Lemma 4.12 we can also extract a the following result about the application of configurators.

Theorem 4.14 (Atomicity)**.** *For all ground composition expressions c, all typing environments Γ, all heaps S, and all objects s, if $\Gamma \vdash c : K \Longrightarrow K'$, Γ types S, and s conforms with K then there is a derivation of $s; c; S \Downarrow s'; S'$.*

Proof. The proof of this theorem follows from the cases of the proof of Lemma 4.12. In particular it follows from the fact that the Rules (Wrong Plug) and (Wrong Uses) are never applicable. □

Combining this with the safety result of Theorem 4.13 we conclude that a configuration process once started is never interrupted due to some run-time error. This result makes precise want we mean by "atomicity" of reconfiguration. Notice that the same result already holds for the language λ_χ^τ of Chapter 3.

This concludes the presentation of our core component calculus with dynamic reconfiguration of objects. Type safety is supported by a combination of dynamic and static typing mechanisms, and ensures at compile-time the atomicity of well-typed reconfiguration actions.

We now present a possible extension of the presented language in order to cope with reconfiguration of objects at an arbitrary depth in a system.

4.3 Reconfiguration at an Arbitrary Depth

The language constructs presented so far address only one level of reconfiguration, i.e. only the top level of an object can be changed in a reconfiguration action. This is due to the uniformity between the expressiveness of reconfiguration and composition using the same composition operations. Compositions are also defined one layer at a time.

However, one may think of adding extra expressive power to our model by adding operations that allow the definition of reconfiguration scripts changing objects more deeply. Take the following example where an example of a new reconfiguration operation, patch r[o], is used to apply a reconfiguration script r to an inner instance of an object o, in a composition context instead of a computational context. (We omitted the majority of the usual type annotations to avoid cluttering the example with unnecessary type expressions.)

```
let C = compose(provides p:I;
                m1[g=fun x→ x];
                m2[f=fun x→ m1.g(x)+1];
                plug m2 into p) in
let D = compose(provides p:I;
                c[C];
                plug c.p into o) in
let o = new D in ...
```

Notice that component D wraps the component C and therefore a λ_ρ reconfiguration script, when applied to an instance of D, can only change the implementation of the internal component c by replacing it completely.

Consider that the semantics of the composition operation patch r[o] is to take a configurator r and applying it to an internal component o. Then, consider the following definition of two configurator values and the corresponding application to an instance of component D, object o in the usual way,

```
let rC = (provides q; plug m2 into q) in
let rD = (provides q; patch rC[c]; plug c.q into q) in
reconfig x = rD[o] in x.q.m1(1) ...
```

Configurator rC, when applied to an instance of C, declares a new provided port q and implements it with the already existent block m2.
We first define configurator rC whose type is

$\{m2 \bullet I\} \Longrightarrow \{m2 \bullet I, q \triangleright I\}$

we then define a configurator rD which takes an inner element of the target object, named c, and applies rC to it using the new operation patch. The patch operation works in a similar way to expression **reconfig** r[o] but it is a composition operation which is evaluated in the context of a composition. So, when configurator rD is applied to object o, it declares a new provided port named q, reconfigures the internal component c, and connects its newly introduced port q to the new port q in object o.

The resource-based language we use to type configurators can be extended to cross the borders of internal components (by allowing resource names with more than 2 levels) and refer to inner elements at any determined depth. According to this idea, the type of rD is

$\{c.m2 \bullet I\} \Longrightarrow \{c.m2 \bullet I, c.q \triangleright I, q \triangleright I\}$.

One interesting aspect of this reconfiguration operation is that a top-level run-time test can still verify the presence of the referred resources and therefore ensure the atomicity of all reconfiguration levels. The semantics presented in this chapter to support dynamic reconfiguration of objects can be extended simply by making the matching test to recursively follow the structure of the names in the hierarchy. Our formulation of the type system and type safety results are expected to hold in this extended reconfiguration language.

4.4 Remarks

In λ_ρ we define a minimal set of composition operations and a type system suitable for capturing properties of both dynamic composition of components and dynamic reconfiguration of objects. We deal with the problem of dynamically discovering the architecture of black-box objects by a structural matching test at run-time that closely relates to a statically ensured property, architectural compliance (Lemma 4.9). The input data for this test is a localised run-time type information in configurator and objects.

However, our approach to reconfiguration is based on some basic assumptions which we argue do not destroy the essence of our model. Here, we informally present these assumptions and explain the implications they have and discuss their orthogonality with relation to the key aspects we have included in our model. Treating these assumptions in full in our core language would clutter the presentation with irrelevant details. It could also, in some cases, compromise the language generality and even the formal soundness proofs as they are written (as it is the case of nil references being part of the source language).

The first assumption made by our model is the unnecessary usage of an explicit deletion operation. An implicit garbage collection discipline, designed to eliminate the elements no longer referred by a provided port, would solve the problem of deleting the "forgotten" elements without disrupting the system's execution. It is possible, in our model, to freely manipulate references to inner elements, exporting them outside the original component boundaries. The inclusion of an explicit elimination operation would imply a tight control over reference passing outside their composition context. Reconfiguration would require reassessing the type checking of the target instance after each reconfiguration step. This highly restricts the generic manipulation of configurators and components as we present it.

Another aspect relates to our forbidding of repetition of names inside composition operations. The type system in λ_{χ}^{τ} disallows this, however, through reconfiguration it is possible to introduce internal elements named with already used names. The new name hides the older one that clashes with it, but because the names are replaced by their locations all the existing structure is maintained. However, hidden names become inaccessible to further reconfigurations.

There are also some design options of our language which imply compromises in the underlying framework. Corrections performed to objects are only valid during the application's lifetime. Once restarted, the original application's code is loaded, and the incorrect code one intended to correct before gets executed again. So, from a software maintenance perspective, a procedure is necessary to apply update schemes on the generating components (the replacement by an up-to-date component). We do not deal with this issue as it is orthogonal to the one we chose to study. The issuer of a reconfiguration script can also issue the corrected components thus completing the diagram in Figure 4.1.

We now relate our work on dynamic reconfiguration with that of other authors.

4.5 Related Work

Our work on dynamic reconfiguration of component instances compares with several approaches by other authors, but nevertheless provides a fresh perspective on the issue of software evolution.

We first compare reconfiguration of instances with the notion of dynamic module replacement in the context of an application. Despite the existence of run-time support for dynamic loading of modules in main-stream frameworks, like COM, CORBA, Java, or .NET, the burden of correctly handling different versions of modules and data in these systems is on the programmer. Fundamental work was developed on type safe mechanisms for the explicit treatment of versioning of modules and abstract data types with type safety [90, 37, 14] to support this kind of software evolution.

Another approach that relates to our work is the one resulting from the use of mixin modules [17, 16, 38, 11, 101, 56, 57, 41] which add to structured module languages, like the ML functor language, the definition of modules by means of composition operations on other modules, allowing also for mutually recursive structures. In the cases of [101, 41] modules are also first-class. These characteristics are primitive in our base calculus λ_{χ}^{τ}, and were already subject of analysis in Section 3.4, we nevertheless recall them here to compare the form of module reconfiguration defined in [41] with our notion of instance reconfiguration.

We also compare our work with others that have introduced programing language mechanisms for supporting dynamic changes of the implementation of objects [35, 36, 89, 91]. We finally comment on the reconfiguration mechanisms on architecture description languages (ADLs) with relation to our work.

Run-Time type information The usage of run-time type information is essential in our language to inspect the structure of objects, which is hidden from typing, and decide at run-time if a reconfiguration action succeeds or not and hence guaranty its atomicity. This is inspired in the widely known dynamic typing and type inspection mechanisms of programming languages notably formalised in [2]. The other side of the coin is the run-time type information in configurators scripts. We see reconfiguration scripts as values that can be imported by programs to correct running objects, which can be (and almost certainly are) developed in different moments. Hence, the run-time type information on configurators, resembles a form of signature or *proof carrying code* [75] in the sense that the reconfiguration is only performed if this information is checked successfully.

Dynamic software update systems. From the perspective of software evolution and dynamic reconfiguration, the works of Sewell [90], Duggan [37] and Bierman et al. [14] all address the operational semantics and type structure of software systems supporting dynamic change of modules implementing certain abstract data types (ADTs), mainly focusing on version management of values of such abstract types.

The model language developed in [90] supports separate compilation and linking of distributed programs, which interact via typed channels, along with a version control mechanism. It allows the definition and execution of new modules while explicitly defining compatibility between abstract types of different modules. Language primitives are provided for defining typed second-class modules, in a language similar to ML functor language, and their implementation, in a distributed programming language similar to the π-calculus, a command language is introduced to define (build) and run modules at a system configuration level. Abstract data types within modules can be made explicitly compatible with prior versions by a language mechanism that forces their actual representation types to be the same. For instance, the sequence of commands:

```
A := (struct type t=trep val x=e main=... end) with! B
run A
```

builds and runs a new program (`A`) based on the module definition expression

```
struct type t=trep val x=e main=... end
```

which defines an abstract type `t` with the internal representation `trep` and a value `x` given by the core expression `e`. The definition `A := ... with! B` forces the equivalence between the shapes of the new module and a previously defined module `B` to be similar and it forces the compatibility between the type representation of `t` and the same type defined in `B`. Build-time checks are made to ensure compatibility between types.

In the case of [37], the equivalence between abstract types is relaxed by the explicit use of isomorphisms between type representations, called version adapters. The calculus defines primitive operations for unifying abstract types in a certain scope (where isomorphisms are defined). Dynamic type information and explicit folding and unfolding operations on modules are then used to determine, depending on the result of a run-time test, the need for an isomorphism application to data values. Type safety ensures that all needed version adapters do exist when well-typed programs are evaluated.

The work of Bierman et al. [14] gives a precise characterisation of a module update mechanism inspired by previous work on dynamic patches in a typed assembly language [55]. It presents a model language, extending the simply typed λ-calculus, with mutually recursive module definitions and an `update` primitive that explicitly defines synchronisation points where a particular module can be upgraded. When an `update` point is reached, the run-time system checks for available updates for the invoking module and atomically performs the replacement. The language provides access to older versions of a given module when explicitly stated, it otherwise implicitly accesses the latest version. To illustrate the calculus we now reproduce a simplified version of an example in [55]. Consider the method bodies m and m' and the module set ms defined below:

$$m \triangleq \{f = \lambda x.\texttt{update in } M.z,\ \ z = 3\}$$
$$m' \triangleq \{z = (5,5)\}$$
$$ms \triangleq \{\texttt{module } M^0 = m\}$$

Module set ms defines the version 0 of a module named M with implementation m. Consider the following program using the module set ms and an expression of the core language:

$$\texttt{modules } ms \texttt{ in } M.f(0)$$

The modules in ms are visible in the enclosed expression $M.f(0)$.

Assuming that there is an update ready to be made to M changing its implementation to m', the program above evaluates as follows:

$$\begin{array}{rl}
 & \texttt{modules}\ ms\ \texttt{in}\ M.f(0) \\
\longrightarrow & \texttt{modules}\ ms\ \texttt{in}\ (\lambda x.\texttt{update in}\ M.z)(0) \\
\longrightarrow & \texttt{modules}\ ms\ \texttt{in}\ \texttt{update in}\ M.z \\
\xrightarrow{M^1=m'} & \texttt{modules}\ ms'\ \texttt{in}\ M.z \\
\longrightarrow & \texttt{modules}\ ms'\ \texttt{in}\ (5,5)
\end{array}$$

where $ms' = ms \cup \{\texttt{module}\ M^1 = m'\}$. When $M.f$ is resolved the latest version of M is M^0 and therefore $M.f$ denotes the abstraction in m. When `update` is executed a new version of M is loaded and the current module set is augmented with M^1 and therefore $M.z$ evaluates to the value of z in the latest version of M. With the presented type system, which is liberal with respect to the signature types of new modules, assigned types are not preserved after a module upgrade.

The latest development on this subject is presented in [93] where, to ensure type safety, is statically verified that from a update point on, the representation of an updated abstract type, is not used in its concrete form by code that relies on the old representation, i.e. code that uses the old versions of modules do not manipulate concrete representations of abstract types. Other than this, a module update must provide value transformer procedures when updating a statically defined data type, must update top-level function definitions using the same type signatures and may freely add new types, functions and variables.

On all the above approaches to dynamic software update the focus is on the compatibility of abstract typed values or on the access to old and new module versions. In our model, components do not usually represent ADTs but rather service providers, and we concentrate on dynamic reconfiguration of architectures, rather than on individual replacement of a module's implementations. On one hand, our calculus has no mechanism for declaring abstract data types and if, in a reconfiguration process, any conversion of data or some state preservation operation is necessary it must be programmed explicitly. On the other hand, the loose coupling provided by interfaces and the strong encapsulation provided by components and objects allow for the modification of the inner structure of objects (their functionality and data representation) while maintaining type integrity.

Module calculi. The *R-calculus* proposed by Fagorzi and Zucca in [41], and later extended by [42] is an evolution of the basic CMS calculus [11] of Ancona and Zucca with first-class modules which according to the authors results in a form of reconfiguration. See Section 3.4 for a description of CMS and other related module calculi. *R-calculus* builds on CMS by defining a notion of interleaved execution of module manipulation operations and core language expressions. The core language in *R-calculus* is the module language itself, hence we can expect that an expressive core language can, in fact, be encoded in *R-calculus*, and indeed computations using first-class modules can be expressed. Notice that [101] shows an encoding of the λ-calculus in *m-calculus*, which is very similar to the *R-calculus*.

The interleaved execution of core and module expressions derives from the selection primitive of CMS, which extracts a component of from a module, a core language expression, to be executed in a execution layer where all module linking operations are completed and produce closed core expressions. In *R-calculus*, a component can be selected, but instead of being extracted from the component, the module enters a linking phase, where it reduces to a closed expression (by substitution of its free names in the context of the module); at this stage a module is called a configuration. The result can then be extracted and used outside the module's context. In this way, the interleaved execution of both kinds of operations allows the manipulation of modules and configurations at run-time, rather than freezing the module structure in a prior linking phase.

A module in *R-calculus*, written $[\iota; o; \rho]$, basically follows the structure of CMS modules where ι maps input component names to variables whose scope are the expression within the module, o maps output component names to expressions (components), and ρ maps local variable names to private components. Besides the basic composition primitives of CMS: **sum**, to combine two modules, **freeze** to resolve deferred input variables by linking them to output component names, and **reduct** to rename both input and output names; *R-calculus* defines a selection primitive **run**, written $e\downarrow_X$, where a module yield by e reduces to a configuration value, written $[\iota; o; \rho \mid e\,]$ where e is the expression associated with the name X within the module. A configuration is an expression in its linking phase, the evaluation of the selected component produces a result that can be, later on, used outside the module's context by means of the operation **result**, written $e\uparrow$. All results in *R-calculus* are either modules or configurations.

For instance, starting from the CMS example in Section 3.4, where e_5 denotes the module

$e_5 = [w \mapsto W; V \mapsto x+y;\ y \mapsto 2, x \mapsto 1+2\,]$

with an input component name W (although not used, w is visible within the module), an output component name V that maps to the component $x+y$, and two local variables x and y. The "execution" of V in e_5, defined by $e_6 \triangleq e_5\downarrow_V$ reduces to the configuration:

$e_6 = [w \mapsto W; V \mapsto x+y;\ y \mapsto 2, x \mapsto 1+2 \mid x+y\,]$

which then evolves in some steps to

$[w \mapsto W; V \mapsto x+y;\ y \mapsto 2, x \mapsto 1+2 \mid 5\,]$

The result of this completed "computation" is then retrieved by $e_6\uparrow$ which yields 5.

Three static type systems are presented in [42] to prevent run-time errors in *R-calculus*. The first type system, strongly inspired on [11], only allows result extraction to happen when there are no more pending inputs on the module, and therefore the whole component expression can be resolved. A second type system makes use of a complex dependency analysis procedure to allow for a more flexible selection of module components. Despite having deferred input variables, a module can be used only if the variables within the selected component refer to already known components or to deferred components that can be made available in some reconfiguration steps. A third proposal relaxes even further the restrictions imposed by the previous type systems, this time allowing for ″missing component″ errors to happen at run-time.

When comparing to the results of [41, 42] we find that in *R-calculus*, where modules are also pure values, it is impossible to define a reconfiguration script in the sense we present in our model, i.e. an expression that modifies a module by referring to its inner (local) elements while maintaining its identity. Hence, it seems impossible to redefine, in place, some functionality in a component by other means than replacing it completely. The reconfiguration mechanism defined in *R-calculus* therefore compares more closely to our notion of dynamic composition, i.e. it produces first-class stateless components out of existing ones, at run-time. We can draw an analogy between the **run** operation and our component construction operation, compose e, which resolves the architecture of a component, and between the **result** operation and our instantiation of components and their subsequent call of the object methods. In addition to this, we also consider the in-place modification of the internal structure of running objects (of local and private elements). Thus, we are able to correct or modify the behaviour of elements (of the same object) whose functionality may indirectly depend on the ones being modified, as a consequence of the reference semantics of the plug composition operator.

At the level of typing, the first of the typing disciplines described above, which imposes that all deferred components are resolved prior to any selection of an inner component, is the one nearest to our type system on the base component language (λ_{χ}^{τ}). The latter approaches are not considered in our study. Besides these similarities we use a combination of dynamic and static type verification when typing reconfiguration actions to match the actual structure of objects against the type of the reconfiguration script. These dynamic checks safely “break” the encapsulation of objects and ensure that all composition operations are applicable to the object, thus allowing for the atomicity of the reconfiguration. This aspect does not seem to be covered by *R-calculus* or any other approach.

Programming language perspective. Forms of dynamic reconfiguration for object-oriented languages involving a fixed predetermined number of future configurations, have been considered in the Java-like language Fickle presented in [35]. The goal of Fickle is to express modifications on the implementation of objects while maintaining their identity. A static and safe type and effect system is provided to ensure type safety. Fickle considers that an object may be reclassified, i.e. that the class that implements it may change during the run of a program but its identity remains the same. This is achieved by a hierarchy of root and state classes allowing for "state" changes to happen by explicit triggers in the program. Consider the example in [36] where the implementation of a video game character is defined as follows:

```
abstract root class Player extends Object {
  bool brave;
  abstract bool wake() {};
  abstract Weapon kissed() { Player };
}
state class Frog extends Player {
  Vocal pouch;
  bool wake() { } {pouch.blow(); return brave;}
  Weapon kissed() { Player } {this⇓Prince; sword:= new Weapon; return sword;}
}
state class Prince extends Player {
  Weapon sword;
  bool wake() {} { sword.swing(); return brave; }
  Weapon kissed() { Player } { return sword; }
  Frog cursed() { Player } {this⇓Frog; pouch:= new Vocal; return this;}
}
```

The root class Player defines here a common structure to all state classes, which is maintained in state changes. The state classes extending the root class define the possible states for an object of class Player. A state change operations, written o⇓C can then be used to change the implementation of object o to state class C. Notice the code in the Frog.kissed method, after the state change (this⇓Prince) it follows as if this was of class Prince. The symmetric happens in the Prince.cursed method.

The following sequence of expressions illustrates the effects of re-classification:

```
Frog mate;
mate.wake();    // inflates pouch
mate.kissed(); // turns into a prince
mate.wake();    // swings sword
```

A type and effect system ensures the type safety of such constructions. A similar idea is presented in [89] but based on dynamic typing mechanisms.

Although modifications may depend on computational results reconfiguration happens in predictable and limited ways. In our work, we aim to model unanticipated reconfiguration using first-class typed notions of (re)configuration scripts, thus following an approach that does not seem to have been explored here.

Another object-oriented programming language which allows for dynamically changing the functionality of objects is Chai [91]. In the definition of a class one can use the standard inheritance mechanism and, additionally, "use" a set of traits (which define behaviour based on a set of expected methods) to extend its base functionality. A trait, used in the definition of a class, can then be replaced, in a particular object, by another, thus changing its implementation. Unlike this trait updating mechanism, our reconfiguration allows, besides the local replacing of elements, the arbitrary addition of new elements and connections to an object's structure.

Evolution on distributed systems Architecture description languages [66, 72, 58, 40, 102] usually describe the behaviour of distributed components at the level of a coordination service. Evolution of the existing sets of connections and components can usually be integrated in the initial description of the system or later introduced into the system and cause changes in the architecture. In general these approaches do not provide mechanisms for predicting and analysing the consequences of reconfiguration scripts or ensuring the consistence of the resulting architectures.

Other approaches operate at system level to reconfigure the architecture of distributed applications based on middleware systems [64, 63, 5, 13] but rely on ad-hoc sequences of API calls and lack any programming language support for the expression of reconfiguration actions.

Chapter 5

Recursion and Polymorphism

In this chapter we study the fundamentals of subtyping second-order equi-recursive types. The results obtained here are afterwards used to enrich our component calculus with new ingredients which are essential in any modern object-oriented language: recursive types and bounded parametric polymorphism.

5.1 Introduction

Recursive types are pervasively used to capture the structure of inductively defined data. They also turn out to be essential to type objects where methods may refer to entities of the self type (see e.g. [1]). On the other hand, polymorphism leverages the flexibility and reusability of software pieces in strongly typed languages, as notably described in [27].

Cardelli and Wegner classify polymorphism, by refining a classification by Strachey [95], as either ad-hoc or universal. Ad-hoc polymorphism is present in most programming languages in the form of implicit coercions and overloading of operators as a practical way of liberating the programmer of many and cumbersome annotations. This happens for instance in the implicit integer and float conversions of arithmetic operations in most languages, or by implicit coercions of objects and primitive values to their textual representation in Java. In the case of universal polymorphism, the classification is subdivided in parametric polymorphism and inclusion polymorphism. Parametric polymorphism denotes the quantification over a range of types allowing for some module or function to be safely applied to different typing contexts. On the other hand, inclusion polymorphism or type inclusion is intrinsically based on the structure of types. It is supported by a universally defined subtyping relation. This is particularly important in most object-oriented languages where it is implicit from using inheritance.

Bounded quantification is a combination of these two kinds of polymorphism (parametric and inclusion) developed in the context of some fundamental languages (e.g. Fun [27], kernel-

Fun, $F_{\leq}$ [34]). In this case the range of type arguments accepted in a parametric module is bounded by subtyping. Thanks to the efforts of a group of researchers, it was incorporated recently into main-stream languages like Java [18] and C♯ [61], thus greatly improving their safety and expressiveness.

Type inclusion denotes a special coercion operation where modifications to the internal representation of the values are many times avoided, like in object-oriented languages, by clever design options. Nevertheless, it can be considered as universal polymorphism because it is supported by a global property, subtyping. This implicit coercion operation is expressed in terms of typing by the following inference rule

$$\text{(Subsumption)}\quad \frac{\Delta \vdash e : \sigma \quad \Delta \vdash \sigma \leq \tau}{\Delta \vdash e : \tau}.$$

This rule intuitively asserts that an expression of a type can be safely used as if it was of any more general type.

In most object-oriented languages, subtyping is intimately related to a nominal class-extension mechanism, and therefore must follow a rigid hierarchy explicitly defined by the programmer. On the other hand, there are languages with richer subtyping relations based on the propagation of basic relations through the structure of type constructors, like type equivalence and general subtyping towards a maximal type $\top$. For instance, the rules

$$\frac{\tau' \leq \tau \quad \sigma \leq \sigma}{\tau \to \sigma \leq \tau' \to \sigma'} \qquad \frac{n \geq m \quad \tau_i \leq \tau_i' \quad \forall_{i \in 1..m}}{\{\ell_i : \tau_i \ ^{i \in 1..n}\} \leq \{\ell_j : \tau_j \ ^{j \in 1..m}\}}$$

denote the relations between the standard type forms for functions and records. The subtyping relation between function types, $\tau \to \sigma$, is based on the subtyping of its subexpressions and expresses the inverted direction in domain set (contravariant in the parameter types) and the same direction in the target set (covariant in the result type). Record types, which are the basis for most object-oriented language encodings [1, 20], are related in two different ways, in depth, by propagating subtyping to field types, and in width, by arbitrary expansion of new fields in subtypes.

When object-oriented programming is at stake, languages like the object calculus of Abadi and Cardelli [1] and Bettini, Bono and Venneri's class and mixins based language, Momi [12], are good examples of how to express general object-oriented programming mechanisms. However, their subtyping relations on classes, objects and mixins stay far behind the flexible relation on records shown above. Both use invariant width subtyping due to unsoundness problems when coding the *self* reference as a generic extra parameter of methods. Abadi and Cardelli present an interesting version of their calculus using variance tags in order to be able to use a richer subtyping relation.

In FJ [59], as well as in Java and C♯, structural equivalence of types, and consequently subtyping, is traded by name-based equivalence, which is very convenient in terms of implementation and also to overcome existing problems with subtyping of recursive types. In fact, structural equivalence of types, although adopted in some experimental programming languages such as OCaml [99] and Modula3 [26], does not seem to have had substantial impact in main-stream object-oriented languages.

Nevertheless, the increasing use of dynamic loading, late binding and mobile code in general purpose programming frameworks raises the issue of finding more flexible compatibility criteria between software components. One reason is that the rigid hierarchy of classes defined by name-based extension and subtyping implies the usage of a global namespace and, for instance, disallows the compatibility of two classes that separately combine the same set of interfaces. This problem can of course be diminished by explicitly using wrapper objects that redirect method calls and therefore make compatible two otherwise incompatible classes. But, structural equivalence would be the most natural solution to this kind of problems. One approach to this problem in main-stream languages is, for instance, Buchi and Weck's work on compound types [21].

Moreover, the task of defining a structure-based subtyping relation is entangled by the inclusion of recursion in the type language. Name-based type systems also use the names as a mean to define recursion and, by using a fixed name hierarchy, do not need to unfold and inspect the structure of types, thus not interfering with subtyping.

Type systems with recursion essentially adopt one of two different forms of relating a recursive type and its unfoldings [3]. They are either related by explicit folding and unfolding language operations, the denominated iso-recursive types, or they are implicitly equivalent according to the equivalence relation,

$$\mu X.\tau = \tau[X \leftarrow \mu X.\tau]$$

where $\mu X.\tau$ indicates that X, when occurring inside τ represents the type itself. This means that, in the limit, $\mu X.\tau$ represents a regular infinite tree of type constructors. Type systems where type equality is interpreted as equality of these infinite trees are called equi-recursive types. We focus our attention on the latter, as solving type equivalence or subtyping using the iso-recursive approach follows a too restrictive syntax-driven subtyping relation.

A pragmatic approach to structural type equivalence is already implemented in Algol68's compiler were record fields are stored and used to find recursive structure on modes (Algol68 terminology for types), the so-called infinite *mode trees* [100]. In Cardelli's Amber language [23], where type inclusion was first introduced, subtyping of equi-recursive types is resolved by a simple inductive rule and axiom

$$\frac{\Delta, X \leq Y \vdash \tau \leq \sigma}{\Delta \vdash \mu X.\tau \leq \mu Y.\sigma} \qquad \Delta, X \leq Y \vdash X \leq Y,$$

which follows the same idea, but addressing type inclusion instead of equivalence. This approach is more powerful than the iso-recursive but it is still restrictive, because it only relates types whose structure is perfectly "synchronised".

Other approaches to the problem of defining an unrestricted subtyping relation between first-order equi-recursive types already exist for quite a while [6, 19, 48]. Also, some work using limited interpretations of type recursivity was developed [51, 28], to test the conservativity of different second-order subtyping relations extended with recursive types, but again using limited relations and with no real intention of covering the whole subtyping relation between equi-recursive types.

Intuitively, when interpreted to the full extent, the subsumption relation between recursive types corresponds to the usual inclusion of infinite (regular) trees, and the difficulty in the polymorphic case arises due to the presence of binding occurrences of type variables on types, introduced by the unfolding of type quantifiers. These subtyping relations have been expressed by means of inductive proof systems, where the coinduction principle appears embedded in various explicit ways. For instance, Amadio and Cardelli [6] define a finite approximation notion on infinite trees. They build and prove correct an algorithm that records a trail of derivations to detect recursion spots. On the other hand, Brandt and Henglein [19] define a special stratified interpretation for sequents that outrules the inconsistency introduced by a fix-point rule

$$\frac{A, P \vdash P}{A \vdash P}$$

where A is the set of assertions and P is the predicate we wish to prove correct. This demands that the proof $A, P \vdash P$ is contractive, i.e. it does not conclude P trivially from A, P. Colazzo and Ghelli [30, 31] have used this kind of rule in their algorithm for second-order types and have used a tagging mechanism to deal with the introduction of type variables to avoid having to perform variable substitution. This tagging mechanism allows the distinction between two different variables that have the same name, an unpleasant consequence is that a true recursion is only discovered if a pair of types is met in the derivation three times.

Our technical approach follows the more natural approach of Gapeyev, Levin and Pierce [48] for first-order types which consists on the definition of the subtyping relation based on the greatest fixed point of a function, also called generating function. For a language with arrow, product and recursive types the authors define a function S on sets of pairs of types as follows

$$\begin{array}{rcl} S(R) & = & \{(\tau,\top) \mid \tau \in \mathcal{T}\} \\ & \cup & \{(\tau \to \sigma, \tau' \to \sigma') \mid (\tau',\tau),(\sigma,\sigma') \in R\} \\ & \cup & \{(\tau \times \sigma, \tau' \times \sigma') \mid (\tau,\tau'),(\sigma,\sigma') \in R\} \\ & \cup & \{(\mu X.\tau,\sigma) \mid (\tau[X \leftarrow \mu X.\tau],\sigma) \in R\} \\ & \cup & \{(\tau,\mu X.\sigma) \mid (\tau,\sigma[X \leftarrow \mu X.\sigma]) \in R\} \end{array}$$

where $\mathcal{T}$ is the set of all type expressions. Then, the subtyping relation is defined as the greatest fixed point of S so that subtyping is given by,

$$\tau \leq \sigma \triangleq (\tau,\sigma) \in \nu S.$$

We define a subtyping relation along the same lines, but introducing mechanisms to deal with the bindings of type variables introduced by polymorphic quantification. After that, we define an algorithm to check the subtyping of two type expressions.

In this chapter, we target the smallest of the theoretical languages mentioned above, the second-order typed lambda calculus with subtyping. The second-order lambda calculus (*F-system* [52]) was introduced independently by Girard and Reynolds and later extended with subtyping and bounded-quantification in Fun [27]. A more general subtyping relation was afterwards presented in [51], the $F_{\leq}$ calculus, but its subtyping relation was later proven undecidable [77, 29]. In this context, we chose to work with kernel-Fun, a generalisation of the Fun calculus introduced in [27]. Here, we present the details of the definitions and corresponding proofs in order to define a generic framework that allows us to reuse it. Although the proof of the closure under transitivity is a bit more involving, we show that all properties of the subtyping relation are proven using standard coinductive techniques. This work applies well to other decidable subtyping disciplines such as $F_{\leq}^{\top}$ [28] as well as to other more complicated languages, such as our component calculus. In Chapter 6, we apply these techniques to our component calculus in order to extend it with polymorphism and type recursion.

5.2 On Subtyping Second-Order Equi-Recursive Types

5.2.1 Subtyping Relation

We start with the type language of kernel-Fun plus type recursion and then present its subtyping relation and some of its properties. Some of these properties are essential to show type safety, others are essential to prove the correctness of our subtyping algorithm.

Definition 5.1 (Types).
The set of types $\mathcal{T}_\lambda$ is defined by the following abstract syntax:

$$\begin{array}{llll} \tau & ::= & & \textit{types} \\ & | & \top & \textit{top} \\ & | & \tau \to \tau & \textit{function type} \\ & | & X & \textit{type variable} \\ & | & \forall_{X \leq \tau}\ \tau & \textit{quantifier type} \\ & | & \mu X.\tau & \textit{recursive type} \end{array}$$

Besides $\top$, the supertype of all types, and the arrow type, both standard in the simply typed lambda calculus, we also have type variables X, Y, Z, etc., quantified types of the form $\forall_{X \leq \tau}\ \tau'$ where X is bound in τ' and all its possible values are subtypes of τ, and recursive types of the form $\mu X.\tau$. The rules we might expect in an inductive subtyping system for this language are the ones in Figure 5.1. These include the usual relationships: maximality of $\top$ (Sub Top), reflexivity (Sub Equal), arrow type inclusion (Sub Fun), transitivity (Sub TransVar), and unfolding of recursive types (Sub RecR) and (Sub RecL). To compare polymorphic types, we adopt a kernel-Fun style rule, (Sub All), where the quantifier bound δ is required to be the same on both types.

Notice that, in Figure 5.1, we do not include the explicit transitivity rule

$$\text{(Sub Trans)} \quad \frac{\Delta \vdash \tau \leq \delta \quad \Delta \vdash \delta \leq \sigma}{\Delta \vdash \tau \leq \sigma}$$

We replace this by Rule (Sub TransVar) as an intermediate step towards a definition of a subtyping algorithm. Notice that δ in Rule (Sub Trans) cannot be determined from the conclusion $\Delta \vdash \tau \leq \sigma$. In the case of kernel-Fun, transitivity can be proven to hold in the structure of types except for type variables, hence subtyping of type variables must be dealt explicitly. Then, we are, in general, able to prove transitivity for the whole relation.

Unfortunately, the adoption of an inductive system based on these rules results in an incomplete type system that does not seem to easily lead to a terminating algorithm, as remarked by Ghelli [51] in the case of $F_{\leq}$. On the other hand, the subtyping algorithm developed by Colazzo and Ghelli in [30] uses these rules together with the already referred fixed point rule of Brandt and Henglein, but again giving sequents and derivations a particular interpretation, far from the usual inductive interpretation. In fact, our difficulties in getting a clear understanding of that work, which is fairly complex, lead us to attempt a different approach, leading to the subtyping relation presented in this chapter.

Our approach is to follow the development of Gapeyev, Levin and Pierce [48] for first-order types, and extend it in an uniform way to polymorphic types. We start from a coinductive definition of the subtyping relation and then prove the properties which are fundamental to type safety in any language such as transitivity, narrowing, weakening, substitution of type variables and equivariance.

$$\text{(Sub Top)}\quad \Delta \vdash \tau \leq \top \qquad \text{(Sub Equal)}\quad \Delta \vdash \tau \leq \tau \qquad \text{(Sub Fun)}\quad \frac{\Delta \vdash \tau' \leq \tau \quad \Delta \vdash \sigma \leq \sigma'}{\Delta \vdash \tau \to \sigma \leq \tau' \to \sigma'}$$

$$\text{(Sub TransVar)}\quad \frac{\Delta \vdash \tau \leq \sigma \qquad X \leq \tau \in \Delta}{\Delta \vdash X \leq \sigma} \qquad \text{(Sub All)}\quad \frac{\Delta, X \leq \delta \vdash \tau \leq \sigma}{\Delta \vdash \forall_{X \leq \delta}\, \tau \leq \forall_{X \leq \delta}\, \sigma}$$

$$\text{(Sub RecL)}\quad \frac{\Delta \vdash \tau[X \leftarrow \mu X.\tau] \leq \sigma}{\Delta \vdash \mu X.\tau \leq \sigma} \qquad \text{(Sub RecR)}\quad \frac{\Delta \vdash \tau \leq \sigma[X \leftarrow \mu X.\sigma]}{\Delta \vdash \tau \leq \mu X.\sigma}$$

Figure 5.1: Subtyping rules.

$$\text{(Env } \phi\text{)}\quad \phi \vdash \diamond \qquad \text{(Env TVar)}\ (X \notin Dom(\Delta))\quad \frac{\Delta \vdash \tau \ \mathsf{ok}}{\Delta, X \leq \tau \vdash \diamond} \qquad \text{(Type Top)}\quad \frac{\Delta \vdash \diamond}{\Delta \vdash \top \ \mathsf{ok}} \qquad \text{(Type Fun)}\quad \frac{\Delta \vdash \tau \ \mathsf{ok} \quad \Delta \vdash \sigma \ \mathsf{ok}}{\Delta \vdash \tau \to \sigma \ \mathsf{ok}}$$

$$\text{(Type TVar)}\quad \frac{\Delta, X \leq \tau \vdash \diamond}{\Delta, X \leq \tau \vdash X \ \mathsf{ok}} \qquad \text{(Type Rec)}\quad \frac{\Delta, X \leq \top \vdash \tau \ \mathsf{ok}}{\Delta \vdash \mu X.\tau \ \mathsf{ok}} \qquad \text{(Type All)}\quad \frac{\Delta \vdash \tau \ \mathsf{ok} \quad \Delta, X \leq \tau \vdash \sigma \ \mathsf{ok}}{\Delta \vdash \forall_{X \leq \tau}\, \sigma}$$

Figure 5.2: Well-formed types and typing environments.

Prior to this, we need to define and prove some basic properties of typing environments, which are then used in the subsequent proofs.

Definition 5.2 (Typing Environments). *The set $\mathcal{D}$ of all valid typing environments is defined by the abstract syntax*

$$\Delta ::= \phi \mid \Delta, X \leq \tau$$

such that $\Delta \vdash \diamond$ is derivable by the rules in Figure 5.2.

We write $\Delta(X)$ to denote the type τ such that $X \leq \tau \in \Delta$, where τ is the bound of X.

Lemma 5.3. *For all $\Delta, \Delta' \in \mathcal{D}$ and $\delta \in \mathcal{T}_\lambda$, If $\Delta \vdash \tau \ \mathsf{ok}$ then $\Delta \vdash \diamond$.*

Lemma 5.4 (Weakening of typing environments). *For all $\Delta, \Delta' \in \mathcal{D}$ and $\delta \in \mathcal{T}_\lambda$,*

1. *if $\Delta, \Delta' \vdash \diamond$, $X \notin Dom(\Delta, \Delta')$, and $\Delta \vdash \delta \ \mathsf{ok}$ then $\Delta, X \leq \delta, \Delta' \vdash \diamond$.*

2. *if $\Delta, \Delta' \vdash \tau \ \mathsf{ok}$, $X \notin Dom(\Delta, \Delta')$, and $\Delta \vdash \delta \ \mathsf{ok}$ then $\Delta, X \leq \delta, \Delta' \vdash \tau \ \mathsf{ok}$.*

Lemma 5.5 (Substitution). *For all* $\Delta, \Delta' \in \mathcal{D}$ *and* $\delta \in \mathcal{T}_\lambda$*,*

1. *If* $\Delta, X \leq \delta, \Delta' \vdash \diamond$ *and* $\Delta \vdash \delta'$ ok *then* $\Delta, \Delta'[X \leftarrow \delta'] \vdash \diamond$.

2. *If* $\Delta, X \leq \delta, \Delta' \vdash \tau$ ok *and* $\Delta \vdash \delta'$ ok *then* $\Delta, \Delta'[X \leftarrow \delta'] \vdash \tau[X \leftarrow \delta']$ ok.

Lemma 5.6 (Variable Exchange). *For all* $\Delta, \Delta', \Delta'' \in \mathcal{D}$ *and* $\delta \in \mathcal{T}_\lambda$*,*

1. *If* $\Delta, X \leq \delta, \Delta', \Delta'' \vdash \diamond$ *and* $X \notin Dom(\Delta')$ *then* $\Delta, \Delta', X \leq \delta, \Delta'' \vdash \diamond$.

2. *If* $\Delta, X \leq \delta, \Delta', \Delta'' \vdash \tau$ ok *and* $X \notin Dom(\Delta')$ *then* $\Delta, \Delta', X \leq \delta, \Delta'' \vdash \tau$ ok.

Lemmas 5.3, 5.4, 5.5, and 5.6 are proven by simple induction on the height of the derivations. We omit the proofs from the core of this dissertation as they are not particularly relevant. Nevertheless, the detailed proof reasoning is shown in Appendix A.3 from page 196 on.

Now we define the set of all subtyping judgements, the elements of $\mathcal{J}$ are tuples (Δ, τ, σ) that denote subtyping judgements of the form $\Delta \vdash \tau \leq \sigma$. We use the notation $(\Delta; \tau; \sigma)$ when commas are needed to express the elements of the tuple.

Definition 5.7 (Subtyping judgements domain). $\mathcal{J} \triangleq \mathcal{D} \times \mathcal{T}_\lambda \times \mathcal{T}_\lambda$

We now define a function S that manipulates sets of tuples according to the rules in Figure 5.1 and then introduce its greatest fixed point as our subtyping relation.

Definition 5.8 (Generating function). *The* generating function *is the map* $S \in \mathcal{P}(\mathcal{J}) \rightarrow \mathcal{P}(\mathcal{J})$ *defined by:*

$$\begin{array}{rcll}
S(\mathfrak{R}) & = & \{(\Delta;\tau;\tau) \mid \Delta \vdash \tau \ \mathsf{ok}\} & \textit{(Sub Equal)} \\
 & \cup & \{(\Delta;\tau;\top) \mid \Delta \vdash \tau \ \mathsf{ok}\} & \textit{(Sub Top)} \\
 & \cup & \{(\Delta;\tau \rightarrow \sigma;\tau' \rightarrow \sigma') \mid (\Delta;\tau';\tau), (\Delta;\sigma;\sigma') \in \mathfrak{R}\} & \textit{(Sub Fun)} \\
 & \cup & \{(\Delta;X;\sigma) \mid X \leq \tau \in \Delta \textit{ and } (\Delta;\tau;\sigma) \in \mathfrak{R}\} & \textit{(Sub TransVar)} \\
 & \cup & \{(\Delta;\forall_{X\leq\delta}\,\tau;\forall_{X\leq\delta}\,\sigma) \mid (\Delta, X \leq \delta;\tau;\sigma) \in \mathfrak{R}\} & \textit{(Sub All)} \\
 & \cup & \{(\Delta;\tau;\mu X.\sigma) \mid (\Delta;\tau;\sigma[X \leftarrow \mu X.\sigma]) \in \mathfrak{R}\ \} & \textit{(Sub RecR)} \\
 & \cup & \{(\Delta;\mu X.\tau;\sigma) \mid (\Delta;\tau[X \leftarrow \mu X.\tau];\sigma) \in \mathfrak{R} \textit{ and } \sigma \not\equiv \mu X.\sigma'\ \} & \textit{(Sub RecL)}
\end{array}$$

By keeping the typing environment in the relation together with the pair of types we keep the correct binding for all type variables occurring in the types. New variables are introduced in the typing environment by the case (Sub All). We assume that X is new in Δ, which is obtained by replacing all occurrences of X in both τ and σ by a fresh type variable. More, notice that the least fixed point of this function is the relation formed by an inductive interpretation of the rules in Figure 5.1.

In order to proceed with the definition of the subtyping relation we verify that S is monotonic, and therefore, confirm that its greatest fixed point $\nu S \in \mathcal{P}(\mathcal{J})$ exists.

Proposition 5.9 (Monotonicity of S). *For all $\mathfrak{R}, \mathfrak{R}' \in \mathcal{P}(\mathcal{J})$, $\mathfrak{R} \subseteq \mathfrak{R}' \Rightarrow S(\mathfrak{R}) \subseteq S(\mathfrak{R}')$.*

Proof Sketch. The proof follows by case analysis on $(\Delta, \tau, \sigma) \in S(\mathfrak{R})$. (for the complete proof see Appendix A.3 on page 198). □

We then define our subtyping relation to be the greatest fixed point of S, noted νS, and define the validity of a subtyping judgement by the membership of its corresponding tuple in the relation.

Definition 5.10 (Subtyping). $\Delta \vdash \tau \leq \sigma \triangleq (\Delta, \tau, \sigma) \in \nu S$.

The relation thus defined enjoys the basic properties of weakening, substitution of type variables, equivariance, narrowing and transitivity, essential to prove the type safety of any underlying language and also the correctness of the subtyping algorithm. In general, these kind of results are proved by somewhat involved inductions on derivations; in our setting, due to the natural definition of subtyping as a greatest fixed point, we handle them by quite standard coinductive proof techniques. Consider the weakening property in νS.

Lemma 5.11 (Weakening). *For all $\Delta, \Delta' \in \mathcal{D}$, and all types $\tau, \sigma, \delta \in \mathcal{T}_\lambda$, if $\Delta, \Delta' \vdash \tau \leq \sigma$, $X \notin Dom(\Delta, \Delta')$, and $\Delta \vdash \delta$ ok then $\Delta, X \leq \delta, \Delta' \vdash \tau \leq \sigma$.*

Proof Sketch. To prove that νS is closed under weakening we consider a set of tuples, $\mathfrak{W}$, built by saturation of all possible weakenings of tuples in νS

$$\mathfrak{W} \triangleq \{(\Delta, X \leq \delta, \Delta'; \tau; \sigma) \mid (\Delta, \Delta'; \tau; \sigma) \in \nu S,\ X \notin Dom(\Delta, \Delta') \text{ and } \Delta \vdash \delta \text{ ok}\}$$

By case analysis in the definition of S we prove $\mathfrak{W}$ to be S-consistent, $\mathfrak{W} \subseteq S(\mathfrak{W})$, and by the coinduction principle we have that $\mathfrak{W} \subseteq \nu S$. (for the complete proof see appendix A.3 on page 199). □

Now, consider transitivity in νS. Transitivity and narrowing are two interdependent properties, and so we must use a slightly different setting that combines both. This combination of narrowing and transitivity already appears in the proofs of transitivity in second-order calculus without recursive types. Now, due to the presence of equi-recursive types we have to consider an even larger relation than usual, including the expected narrowing and transitivity relations.

We start by defining the notion of narrowing on a typing environment, and then define the closure of νS under narrowing.

Definition 5.12 (Narrowing). *For all $\Delta, \Delta' \in \mathcal{D}$, we have that Δ is narrower than Δ' with relation to $\mathfrak{R} \in \mathcal{P}(\mathcal{J})$, written $\Delta \sqsubseteq_{\mathfrak{R}} \Delta'$, where the relation $\sqsubseteq_{\mathfrak{R}}$ is inductively defined by:*

$$\begin{array}{l} \emptyset \sqsubseteq_{\mathfrak{R}} \emptyset \\ \Gamma, X \leq \gamma \sqsubseteq_{\mathfrak{R}} \Gamma', X \leq \gamma' \quad \textit{if } \Gamma \sqsubseteq_{\mathfrak{R}} \Gamma' \textit{ and } (\Gamma', \gamma, \gamma') \in \mathfrak{R}. \end{array}$$

When a typing environment is narrowed more than once we write $\Delta \sqsubseteq^{n}_{\mathfrak{R}_i \; i \in 1..n} \Gamma$ to denote the sequence of related typing environments $\Delta \sqsubseteq_{\mathfrak{R}_1} \Gamma_1 \sqsubseteq_{\mathfrak{R}_2} \cdots \sqsubseteq_{\mathfrak{R}_n} \Gamma_n$ and $\Gamma_n = \Gamma$.

Definition 5.13 (Closure of νS under narrowing). *For all $n \in \mathbb{N}$ we inductively define $\mathfrak{N}^n$ by:*

$$\begin{array}{rl} \mathfrak{N}^0 \triangleq & \nu S \\ \mathfrak{N}^n \triangleq & \{(\Delta, \tau, \sigma) \mid (\Gamma, \tau, \sigma) \in \mathfrak{N}^{n-1} \textit{ and } \Delta \sqsubseteq_{\mathfrak{N}^{n-1}} \Gamma\}. \end{array}$$

Proposition 5.14 (νS is closed under exchange). *For all $\Delta, \Delta', \Delta'' \in \mathcal{D}$ and $\tau, \sigma, \delta \in \mathcal{T}$, If $(\Delta, X \leq \delta, \Delta', \Delta''; \tau; \sigma) \in \nu S$ and $X \notin FV(\Delta')$ then $(\Delta, \Delta', X \leq \delta, \Delta''; \tau; \sigma) \in \nu S$.*

Proof Sketch. We use again the same coinductive technique this time based on the auxiliary set $\mathfrak{P}$ defined by:

$$\begin{array}{rl} \mathfrak{P} \triangleq & \{(\emptyset, \tau, \sigma) | (\emptyset, \tau, \sigma) \in \nu S\} \\ \cup & \{(\Delta, \Delta', X \leq \delta, \Delta''; \tau; \sigma) \mid (\Delta, X \leq \delta, \Delta', \Delta''; \tau; \sigma) \in \nu S \text{ and } X \notin FV(\Delta')\}. \end{array}$$

Notice that $\nu S \subseteq \mathfrak{P}$. All tuples in νS with empty environments are in $\mathfrak{P}$ by the first case of the definition, and the remaining tuples of νS are in $\mathfrak{P}$ with $\Delta' = \emptyset$ in the second case.

We then prove that $\mathfrak{P}$ is *S-consistent* by picking a tuple $t \in \mathfrak{P}$ and the corresponding tuple $t' \in \nu S$. We analyse all possible cases of t' being supported in νS, and in all cases we reach the conclusion that $t \in S(\mathfrak{P})$. Thus, by the coinduction principle we conclude that $\mathfrak{P} \subseteq \nu S$. (for the complete proof see appendix A.3 on page 200). □

We now define a new relation, larger than the closure of νS under transitivity and narrowing, depicted by the derivation

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\Gamma^n_0 \vdash \tau \leq \alpha_1}}{\vdots}}{\Delta \vdash \tau \leq \alpha_1} \quad \cdots \quad \dfrac{\dfrac{\dfrac{\vdots}{\Gamma^n_{n-1} \vdash \alpha_{n-1} \leq \sigma}}{\vdots}}{\Delta \vdash \alpha_{n-1} \leq \sigma}}{\Delta \vdash \tau \leq \sigma}$$

where $\Delta \vdash \tau \leq \alpha_1$ is the result of n narrowings of $\Gamma^n_0 \vdash \tau \leq \alpha_1$. The latter is supported contractively in the relation, i.e. it is not the result of another explicit narrowing. The same is true for $\Delta \vdash \alpha_{n-1} \leq \sigma$ and all the intermediate premises. The relation is thus defined by

Definition 5.15 (Extended Transitive Closure of νS)**.**

$$\mathfrak{T} \triangleq \{(\Delta, \alpha_0, \alpha_n) \mid \exists_{n\in\mathbb{N}}.\exists_{\alpha_0..\alpha_n\in\mathcal{T}_\lambda}.\forall_{i\in 1..n}.(\Delta, \alpha_{i-1}, \alpha_i) \in \mathfrak{N}^n \}$$

Notice that this relation includes the closure of νS under the standard formulation of transitivity (with $n = 2$) and also the closure under narrowing (with $n = 1$).

Lemma 5.16 (νS is closed under transitivity)**.** *$\mathfrak{T} \subseteq \nu S$.*

Proof Sketch. The proof is done by applying the coinduction principle, showing that $\mathfrak{T}$ is S-*consistent*, i.e. $\mathfrak{T} \subseteq S(\mathfrak{T})$, and by the coinduction principle conclude that $\mathfrak{T} \subseteq \nu S$. We use an internal induction on the number of tuples of $\mathfrak{N}^n$ that form a chain from α_0 to α_n. There are a certain number of cases where a chain of n tuples can be reconstructed in a chain of $n-1$ tuples and the induction hypothesis applies. In the other cases we prove that this tuple is in $\mathfrak{T}$ by means of a different chain of tuples. The cases of interest are the ones involving the transitive case for type variables (Sub TransVar) both in the base of the induction and in the inductive case. In the proof, we write $\mathfrak{T}^i$ to denote the subset of $\mathfrak{T}$ containing only the tuples supported by chains of size i.

Case: $n = 1$ **and** $\alpha_0 = X$ **(Sub TransVar)**

This is the subcase of the induction base $n = 1$ where $(\Delta, X, \alpha_1) \in \mathfrak{T}$ is supported by narrowing by $(\Gamma, X, \alpha_1) \in \nu S$ with $\Delta \sqsubseteq_{\nu S} \Gamma$ and in turn by $(\Gamma, \Gamma(X), \alpha_1) \in \nu S$. By definition of $\mathfrak{N}^1$ (Definition 5.12) we also know that $(\Gamma, \Delta(X), \Gamma(X)) \in \nu S$ and therefore $(\Delta, \Delta(X), \Gamma(X))$, $(\Delta, \Gamma(X), \alpha_1) \in \mathfrak{N}^1$. Which means that, by definition of $\mathfrak{T}$, $(\Delta, \Delta(X), \alpha_1) \in \mathfrak{T}$ (in $\mathfrak{T}^2$). Finally, by the case (Sub TransVar) in the definition of S (Definition 5.8), we conclude that $(\Delta, X, \alpha_1) \in S(\mathfrak{T})$ by (Sub TransVar).

This larger chain of tuples that supports the proof justifies the absence of an inductive solution for the problem, only possible using coinductive proof techniques.

Subcase: $n > 1$ **and** $\alpha_0 = X$ **(Sub TransVar)**

In the inductive case, by the definitions of $\mathfrak{N}^n$ and νS we know that the tuple we are focusing on, $(\Delta, X, \alpha_1) \in \mathfrak{N}^n$, is supported by $(\Gamma^n, X, \alpha_1) \in \nu S$ which in turn is supported by $(\Gamma^n, \Gamma^n(X), \alpha_1) \in \nu S$, case (Sub TransVar). This implies by definition that $(\Delta, \Gamma^n(X), \alpha_1) \in \mathfrak{N}^n$. Also, by definition of $\mathfrak{N}^n$ (Definition 5.12), we know that there is a sequence of tuples supporting the narrowings of the first tuple:

$$(\Gamma^1, \Delta(X), \Gamma^1(X)) \in \mathfrak{N}^{n-1}, (\Gamma^2, \Gamma^1(X), \Gamma^2(X)) \in \mathfrak{N}^{n-2}, \cdots, (\Gamma^n, \Gamma^{n-1}(X), \Gamma^n(X)) \in \mathfrak{N}^0.$$

All these typing environments can all be narrowed to Δ preserving the relation between types. Thus, by definition of $\mathfrak{N}^n$ we conclude that there is a sequence of n tuples in $\mathfrak{N}^n$:

$$(\Delta, \Delta(X), \Gamma^1(X)) \in \mathfrak{N}^n, (\Delta, \Gamma^1(X), \Gamma^2(X)) \in \mathfrak{N}^n, \cdots, (\Delta, \Gamma^{n-1}(X), \Gamma^n(X)) \in \mathfrak{N}^n.$$

Another chain of size n in $\mathfrak{N}^n$ comprises the tuples starting with the bound of X to α_n:

$$(\Delta, \Gamma^n(X), \alpha_1), (\Delta, \alpha_1, \alpha_2), \cdots, (\Delta, \alpha_{n-1}, \alpha_n).$$

So, we conclude that $(\Delta, \Delta(X), \alpha_n) \in \mathfrak{T}$ (in $\mathfrak{T}^{2n}$) and therefore $(\Delta, X, \alpha_n) \in S(\mathfrak{T})$. Again the solution lies on a larger chain of tuples, which still belongs to the greatest fixed point of S.

We prove, in all cases, that $\mathfrak{T}$ is *S-consistent* and therefore, by the coinduction principle, we conclude that $\mathfrak{T} \subseteq \nu S$. (for the complete proof see appendix A.3 on page 201). □

Proposition 5.17 (Transitivity). *For all $\Delta \in \mathcal{D}$ and $\tau, \delta, \sigma \in \mathcal{T}_\lambda$, If $\Delta \vdash \tau \leq \delta$ and $\Delta \vdash \delta \leq \sigma$ then $\Delta \vdash \tau \leq \sigma$.*

Proof. This is a corollary of Lemma 5.16. If $\Delta \vdash \tau \leq \delta$ and $\Delta \vdash \delta \leq \sigma$ then by Definition 5.10 we know that (Δ, τ, δ), $(\Delta, \delta, \sigma) \in \nu S$ and therefore in $\mathfrak{N}^2$. By definition of $\mathfrak{T}$ (Definition 5.15) and Lemma 5.16, we have $(\Delta, \tau, \sigma) \in \nu S$ and therefore $\Delta \vdash \tau \leq \sigma$. □

Moreover, substitution of type variables in subtyping judgements is sound, and the subtyping relation is closed under name permutation. This is proven using the same coinductive technique.

Lemma 5.18 (Substitution of type variables). *For all $\Delta, \Delta' \in \mathcal{D}$, and $\tau, \sigma, \delta, \delta' \in \mathcal{T}_\lambda$, if $\Delta, X \leq \delta, \Delta' \vdash \tau \leq \sigma$ and $\Delta \vdash \delta' \leq \delta$ then we have $\Delta, \Delta'[X \leftarrow \delta'] \vdash \tau[X \leftarrow \delta'] \leq \sigma[X \leftarrow \delta']$.*

Proof Sketch. The proof follows a similar reasoning this time with the set

$$\mathfrak{Q} \triangleq \nu S \cup \{(\Delta, \Delta'[X \leftarrow \delta']; \tau[X \leftarrow \delta']; \tau'[X \leftarrow \delta']) \mid (\Delta, X \leq \delta', \Delta'; \tau; \sigma), (\Delta, \delta', \delta) \in \nu S\}$$

which we use to prove that νS is closed under the substitution of type variables. (for the complete proof see Appendix A.3 on page 204). □

Lemma 5.19 (Equivariance). *For all $\Delta \in \mathcal{P}(\mathcal{J})$, $\tau, \sigma \in \mathcal{T}_\lambda$, if $\Delta \vdash \tau \leq \sigma$ then $\Delta[X \leftrightarrow Y] \vdash \tau[X \leftrightarrow Y] \leq \sigma[X \leftrightarrow Y]$.*

Proof Sketch. This property is proven by means of weakening and substitution of type variables. We transform the replacing of type variables $[X \leftrightarrow Y]$ by a triplet of substitutions $[X \leftarrow Z][Y \leftarrow X][Z \leftarrow Y]$ where Z is fresh.(for the complete proof see Appendix A.3 on page 205). □

So far we have defined and analysed a subtyping relation for kernel-Fun. The next section presents an algorithm that checks whether a given subtyping judgement is in the relation defined here.

5.2.2 Subtyping Algorithm

Decidability of the typing relation depends on the existence of an algorithm to determine the type of an expression, or simply to check if a given typing judgement is valid or not. As expected, the subsumption rule, which usually uses the subtyping relation, cannot be used literally in the typing algorithm because we cannot deterministically choose the resulting supertype from the input, the typing environment and the expression. The implementation of a typing algorithm usually replaces the subsumption rule by subtyping verifications in other rules. The subtyping relation is used only for testing the validity of judgements of the form $\Delta \vdash \tau \leq \sigma$.

In this context, we present and prove correct an algorithm for deciding the validity of a subtyping judgement with respect to the relation in Definition 5.10. The technique depicted here is then used in the development of the typing algorithm for the component language presented in the next chapter. Our algorithm closely follows existing algorithms for first-order equirecursive types [6, 19, 48]. Briefly, these algorithms progress by computing, given a pair of types to be checked for subsumption, a consistent set of pairs that includes it. By the coinduction principle, all the pairs in the set belong to the greatest fixed point. The consistent set is built by saturating the current approximation through backward rule application, and accumulating all supporting pairs of types, until a terminal case, corresponding to the application of an axiom, is found, or an already visited pair is encountered and a cycle is established in the derivation. This corresponds to an unfolding step where both types agree.

In line with our definition, we naturally extend those approaches by building on the generating function in Definition 5.8 and defining a membership check procedure for a tuple $t \in \mathcal{J}$ in the subtyping relation νS. To maintain the binding occurrences of type variables our algorithm manipulates entire judgements instead of pairs of types; this turns out to lead to a remarkably simple way of dealing with type variables. Notice that environments grow as a result of comparing polymorphic types, and, due to α-equivalence, the greatest fixed point νS is closed under renaming (Lemma 5.19). Moreover, we prove that, in our setting, the number of tuples reachable from a given starting tuple are finite up to such renaming and pruning of useless variables (Lemma 5.33). Therefore, our algorithm checks for membership of a tuple in the current approximation modulo a similarity relation on tuples that includes renaming. This allows us to detect cycles at the level of equivalence classes based on similarity, instead of expecting the exact tuple to reappear in the successive approximations. We define similarity of tuples as follows:

$$\begin{tikzcd}
(\Delta,\tau,\sigma) \arrow[r, phantom, "\simeq"] \arrow[d, "\text{substitution}"'] & (\Delta',\tau',\sigma') \\
(\Gamma,\tau,\sigma) \arrow[r, "\rho", "\text{equivariance}"'] & (\Gamma',\tau',\sigma') \arrow[u, "\text{weakening}"']
\end{tikzcd}$$

Figure 5.3: Closure of νS under similarity.

Definition 5.20 (Similarity). Similarity *is the binary relation* $\simeq$ *on* $\mathcal{J}$ *defined by:* $(\Delta,\tau,\sigma) \simeq (\Delta',\tau',\sigma')$*, if there are two typing environments* $\Gamma \subseteq \Delta$ *and* $\Gamma' \subseteq \Delta'$ *with* $\Gamma \vdash \tau\ \mathsf{ok}$*,* $\Gamma \vdash \sigma\ \mathsf{ok}$*, and* $\Gamma' \vdash \tau'\ \mathsf{ok}$*,* $\Gamma' \vdash \sigma'\ \mathsf{ok}$*, and a bijection* $\rho : Dom(\Gamma) \to Dom(\Gamma')$ *such that* $\rho(\Gamma) = \Gamma'$*,* $\rho(\tau) = \tau'$*,* $\rho(\sigma) = \sigma'$*.*

In the sequel, and in particular when defining the subtyping algorithm, we will use the following abbreviation $t \in^{\simeq} A \triangleq \exists u \in A.t \simeq u$. Notice that similarity is decidable, it can be checked by matching the structure of types, modulo bijective renaming of their free type variables, recursively checking if the corresponding bounds are similar. All unused variables are discarded from this comparison as they are not included in the minimal common typing environment supporting the similarity relation.

Example 5.21. For instance, consider the tuples

$$\begin{aligned} t &= (X \le \tau \to \tau, Y \le X\ ;\ \mu Z.X \to Z\ ;\ X) \\ t' &= (Z \le \tau \to \tau \qquad\qquad ;\ \mu X.Z \to X\ ;\ Z) \end{aligned}$$

We know that $t \simeq t'$ because there is a $\Gamma = X \le \tau \to \tau$, $\Gamma' = Z \le \tau \to \tau$ and $\rho = [X \leftrightarrow Z]$ such that $\rho(\Gamma) = \Gamma'$, $\rho(\mu Z.X \to Z) = \mu X.Z \to X$ and $\rho(X) = Z$.

Nevertheless, this can be mechanically determined by inspecting the structure of types and maintaining a name correspondence between the two tuples. In this case the comparison of $\mu Z.X \to Z$ and $\mu X.Z \to X$ leads directly to the bijection $[X \leftrightarrow Z]$ as X and Z are free variables in corresponding places. X and Z have similar bounds, in this case equivalent, and they agree on the second type of the tuple. Notice that Y is redundant in t and is ignored by this mechanical verification.

Another important fact about the subtyping relation is that νS is closed under similarity. This allows us to see the subtyping relation as a set of equivalence classes on tuples.

Definition 5.22 (Closure under similarity). *For any* $R \in \mathcal{P}(\mathcal{J})$ *we define the closure of* R *under similarity, noted* R^**, by* $R^* \triangleq \{t' \mid t \in R \text{ and } t' \simeq t\}$*.*

As an abbreviation, given a tuple $t \in \mathcal{J}$ we write t^* for $\{t\}^*$.

```
$Subtyping(A, (\Delta, \tau, \sigma)) =$
    if $(\Delta, \tau, \sigma) \in^{\simeq} A$ then $A$
    else let $A_0 = A \cup \{(\Delta, \tau, \sigma)\}$ in
    if $\tau \equiv \sigma$ then $A$
    else if $\sigma \equiv \top$ then $A$
    else if $\tau \equiv \tau' \rightarrow \tau''$ and $\sigma \equiv \sigma' \rightarrow \sigma''$ then
        let $A_1 = Subtyping(A_0, (\Delta, \tau'', \sigma''))$ in $Subtyping(A_1, (\Delta, \sigma', \tau'))$
    else if $\tau \equiv X$ then $Subtyping(A_0, (\Delta, \Delta(X), \sigma))$
    else if $\tau \equiv \forall_{X \leq \delta} \tau'$ and $\sigma \equiv \forall_{X \leq \delta} \sigma'$ then $Subtyping(A_0, (\Delta, X \leq \delta; \tau'; \sigma'))$
    else if $\tau \equiv \mu X.\tau'$ then $Subtyping(A_0, (\Delta, \tau'[X \leftarrow \tau], \sigma))$
    else if $\sigma \equiv \mu X.\sigma'$ then $Subtyping(A_0, (\Delta, \tau, \sigma'[X \leftarrow \sigma]))$
    else $fail$
```

Figure 5.4: Subtyping algorithm.

Lemma 5.23 (νS is closed under similarity)**.** $\nu S^* = \nu S$.

Proof. By reflexivity of similarity we know that $\nu S \subseteq \nu S^*$. We prove that the opposite is also true based on existing properties of the subtyping relation that relate similar tuples in the way depicted in Figure 5.3. For any tuple $t' \in \nu S^*$, we want to show that $t' \in \nu S$. By Definition 5.22, we have a $t \in \nu S$ such that $t' \simeq t$. Let $t = (\Delta, \tau, \sigma)$ and $t' = (\Delta', \tau', \sigma')$. By Definition 5.20 there are Γ and Γ' such that $\Gamma \vdash \tau$ ok, $\Gamma \vdash \sigma$ ok, $\Gamma' \vdash \tau'$ ok, and $\Gamma' \vdash \sigma'$ ok. There is also a bijection $\rho : Dom(\Gamma) \rightarrow Dom(\Gamma')$ such that $\rho(\Gamma) = \Gamma'$, $\rho(\tau) = \tau'$, and $\rho(\sigma) = \sigma'$. By applying the substitution lemma (Lemma 5.18) on t, to replace each variable $X \in Dom(\Delta \backslash \Gamma)$ in t by its bound $\Delta(X)$. Since such type variables do not occur in Γ, τ and σ, we conclude that $(\Gamma, \tau, \sigma) \in \nu S$. By equivariance (Lemma 5.19), we conclude that $(\Gamma', \tau', \sigma') \in \nu S$. By weakening (Proposition 5.11) we conclude that $t' \in \nu S$. Hence $\nu S^* \subseteq \nu S$. □

We now have the necessary ingredients to define our algorithm. The procedure *Subtyping* defined next, when called with an empty set of assumptions (tuples) and a tuple (Δ, τ, σ), $Subtyping(\emptyset, (\Delta, \tau, \sigma))$, corresponds to checking the judgement $\Delta \vdash \tau \leq \sigma$. The result of this procedure is a consistent set of tuples that support the initial argument and by the coinduction principle it implies its inclusion in the greatest fixed point. In the event of failure, the judgement is not valid.

Definition 5.24 (Subtyping algorithm)**.** *$Subtyping(A, (\Delta, \tau, \sigma))$ is defined by the procedure described in Figure 5.4.*

Notice that the terminal cases of the algorithm correspond either to a self-supported case of the relation, (Sub Equal) and (Sub Top), to a cycle detection by repetition of a tuple in A, or to a mismatch of the type shapes (the alternative to all other cases). In the latter, the result $fail$

automatically propagates through the entire call stack like an exception being raised at that point. The remaining cases recur on the structure of types and in a larger set of tuples, A_0, where (Δ, τ, σ) is included. Thus, if, due to some recursive type, (Δ, τ, σ) is encountered again in the derivation, the membership test $(\Delta, \tau, \sigma) \in^{\simeq} A$ succeeds and that particular branch of the algorithm succeeds.

Remark 5.25. Notice that, with relation to the standard first-order subtyping algorithm of [48, 78], the fundamental differences are concentrated on the domain of the function, which includes the context of type variables, and particularly on the halting condition, which works modulo the similarity relation defined in Definition 5.20. This allows for a better intuitive understanding about the obtained result and it provides a better technical support for proving its correctness.

Next, we prove that the subtyping algorithm yields correct results. We start by showing that the algorithm terminates on all inputs and, that upon termination the result is correct. We show that the search space of the algorithm, comprising all tuples reachable from the initial argument, is finite modulo the similarity relation presented above. To characterise such search space we first introduce some notions about subexpressions, type variable scoping and reachability.

Definition 5.26 (Subexpression). Subexpression *is the binary relation on types, written $\tau \preceq \sigma$, inductively defined as follows:*

$$\begin{aligned}
\tau[X \leftarrow \mu X.\tau] &\preceq \mu X.\tau \\
\tau &\preceq \tau \rightarrow \sigma \\
\sigma &\preceq \tau \rightarrow \sigma \\
\delta &\preceq \forall_{X \leq \delta}\ \tau \\
\tau[X \leftarrow Y] &\preceq \forall_{X \leq \delta}\ \tau \text{ (with } Y \text{ a fresh variable)} \\
\tau \preceq \sigma' &\text{ and } \sigma' \preceq \sigma \text{ (for some type } \sigma'\text{).}
\end{aligned}$$

Definition 5.27 (Reachability). *Reachability is the binary relation on $\mathcal{J}$, noted $t \gg t'$, inductively defined by:*

1. $(\Delta, \tau, \sigma) \gg (\Delta, \tau, \sigma)$
2. *if* $(\Delta, \tau, \sigma) \gg (\Delta', X, \sigma')$ *then* $(\Delta, \tau, \sigma) \gg (\Delta', \Delta'(X), \sigma')$
3. *if* $(\Delta, \tau, \sigma) \gg (\Delta', \forall_{X \leq \delta}\ \tau; \forall_{X \leq \delta}\ \tau')$ *then* $(\Delta, \tau, \sigma) \gg (\Delta', X \leq \delta; \tau; \tau')$
4. *if* $(\Delta, \tau, \sigma) \gg (\Delta'; \tau; \mu X.\sigma)$ *then* $(\Delta, \tau, \sigma) \gg (\Delta'; \tau; \sigma[X \leftarrow \mu X.\sigma])$
5. *if* $(\Delta, \tau, \sigma) \gg (\Delta'; \mu X.\tau; \sigma)$ *then* $(\Delta, \tau, \sigma) \gg (\Delta'; \tau[X \leftarrow \mu X.\tau]; \sigma)$
6. *if* $(\Delta, \tau, \sigma) \gg (\Delta'; \tau' \rightarrow \tau''; \sigma' \rightarrow \sigma'')$
   then $(\Delta, \tau, \sigma) \gg (\Delta'; \sigma'; \tau')$ *and* $(\Delta, \tau, \sigma) \gg (\Delta'; \tau''; \sigma'')$

For any $t \in \mathcal{J}$ we use the abbreviation $Reach(t) \triangleq \{t' \mid t \gg t'\}$.

Lemma 5.28. *If $t' \in Reach(t)$ then all types occurring in t' are subexpressions of types occurring in t.*

Proof. By induction on the notion of Reachability.

1. If $t' = t$ then $t \gg t'$.

2. If $t \gg (\Delta', X, \sigma')$ then by induction hypothesis, (Δ', X, σ') contains only subexpressions of the types occurring in t. Since $\Delta(X)$ is a type expression occurring in t we conclude that all types occurring in $(\Delta, \Delta(X), \sigma')$ are subexpressions of types occurring in t.

3. If $t \gg t''$ with $t'' = (\Delta', \forall_{X \leq \delta}\, \tau; \forall_{X \leq \delta}\, \tau')$, by induction hypothesis t'' contains only subexpressions of the types occurring in t. By Definition 5.26 the types occurring in $(\Delta', X \leq \delta; \tau; \tau')$ are subexpressions of the types occurring in t'' and hence, by transitivity, in t.

4. If $t \gg t''$ with $t'' = (\Delta'; \tau; \mu X.\sigma)$ then by induction hypothesis t'' contains only subexpressions of t. By Definition 5.26 the types contained in $(\Delta'; \tau; \sigma[X \leftarrow \mu X.\sigma])$ are subexpressions of the types occurring in t'', and hence, by transitivity, in t.

5. If $t \gg t''$ with $t'' = (\Delta'; \mu X.\tau; \sigma)$ the reasoning is similar to the case 4.

6. If $t \gg t''$, and $t'' = (\Delta'; \tau \to \sigma; \tau' \to \sigma')$ then by induction hypothesis it contains only subexpressions of types in t. By Definition 5.26 the types in $(\Delta'; \tau'; \tau)$ and $(\Delta'; \sigma; \sigma')$ are subexpressions of the types in occurring in t'' and by transitivity in t.

□

Definition 5.29 (Variable chain)**.** *We define* a chain for a type variable X_0 in τ *to be the sequence of occurrences of type variables $X_0, X_1, \ldots, X_n$ in the type τ such that τ can be expressed in the form*

$$\tau = C_0[\forall_{X_0 \leq \delta_0}\, C_1[\forall_{X_1 \leq \delta_1}\, C_2[\forall_{X_2 \leq \delta_2}\, \ldots]]]$$

where for any $i \geq 0$, X_i may occur in any δ_j with $j > i$, and where $C_i[\ldots]$ are syntactical type contexts.

Intuitively, a variable chain is a sequence of type variables, each one occurring in the bound of the next one. Note that each type variable in a type may have multiple variable chains, i.e. several different ways of writing the type according to the definition. Hereafter, when referring to variable chains in a subtyping judgement $\Delta \vdash \tau \leq \sigma$, the type variables in Δ are implicitly quantified over the type expressions τ and σ. We write $\forall_\Delta \tau$ to denote the type expression obtained by quantifying all variables of Δ over the type expression τ. This notation makes the notion of variable chain uniform both in types and tuples and establishes a common ground to reason about the possible evolutions of the algorithm. We now define some properties on variable chains:

Definition 5.30 (Maximal chain). *Let $\tau \in \mathcal{T}_\Lambda$, we define $\|\tau\|$ to be the length of any maximal chain in τ. For a tuple $t \in \mathcal{J}$, we let $\|t\| \triangleq max(\|\forall_\Delta \tau\|, \|\forall_\Delta \sigma\|)$ where $t = (\Delta, \tau, \sigma)$.*

This notion of length of maximal chains establishes a measure that allows for the analysis of the length of all the chains that appear in a run of the algorithm. The next lemma is particularly useful to ensure this property.

Lemma 5.31 (Preservation of maximal length). *For all $\tau, \sigma, X \in \mathcal{T}_\Lambda$, $\|\sigma[X \leftarrow \tau]\| \leq max(\|\sigma\|, \|\tau\|)$.*

Proof. Consider a maximal chain of $\sigma[X \leftarrow \tau]$. There are three possible ways of characterising this chain. It is either an original chain of σ, an original chain of τ, or it is a chain that starts with a variable of σ and at some point switches to a variable of τ or vice-versa. The two first cases immediately lead to the conclusion. In the third case, it must be the case that a variable declared in σ is in a bound of τ, or vice-versa. This is impossible because the scope of such variables is a subexpression of σ and the type variable substitution avoids the capture of local variables. □

Finally, we enunciate a lemma that limits the length of a maximal chain within the progress of the algorithm.

Lemma 5.32. *For all $t, t' \in \mathcal{J}$, if $t \gg t'$ then $\|t'\| \leq \|t\|$.*

Proof. By induction on the notion of Reachability.

1. If $t' = t$ then $\|t'\| = \|t\|$.

2. If $t \gg (\Delta', X, \sigma')$ then by induction hypothesis $\| (\Delta', X, \sigma') \| \leq \| t \|$. Now, we have two possibilities to locate a maximal chain in $t' = (\Delta', \Delta'(X), \sigma')$: it can be in $\forall_{\Delta'} \Delta'(X)$, in $\forall_{\Delta'} \sigma'$ which are also chains of Δ'. In all cases $\|t'\| \leq \|t\|$.

3. If $t \gg t''$ with $t'' = (\Delta'; \forall_{X \leq \delta}\ \tau; \forall_{X \leq \delta}\ \tau')$, then by the induction hypothesis $\|t''\| \leq \|t\|$. In this case there are two possibilities for the maximal chains of $t' = (\Delta', X \leq \delta; \tau; \tau')$: either a maximal chain involves X and by Definition 5.29 the contribution of X and its bound to the length of a maximal chain is the same, or no maximal chain involves X and the maximal chain in $\forall_{\Delta', X \leq \delta} \tau$ and $\forall_{\Delta', X \leq \delta} \tau'$ is the same. In both cases the maximal chains have the same length, so $\|t'\| \leq \|t\|$.

4. If $t \gg t''$ with $t'' = (\Delta'; \tau; \mu X.\sigma)$ then by induction hypothesis $\|t''\| \leq \|t\|$. Now, consider $t' = (\Delta'; \tau; \sigma[X \leftarrow \mu X.\sigma])$. If all maximal chains of t'' are in Δ' and τ, then all maximal chains of t' are also in Δ' and τ and $\|t'\| \leq \|t\|$. If, on the other hand, one maximal chain is in Δ' and $\mu X.\sigma$ then we have by Lemma 5.31, $\|\sigma[X \leftarrow \mu X.\sigma]\| \leq max(\|\sigma\|, \|\mu X.\sigma\|)$. Hence $\|t'\| \leq \|t\|$.

5. The case where $t \gg t''$ with $t'' = (\Delta'; \mu X.\tau; \sigma)$ is similar to 4.

6. If $t \gg t''$, $t'' = (\Delta'; \tau \rightarrow \sigma; \tau' \rightarrow \sigma')$ then by induction hypothesis we have that $\|t''\| \leq \|t\|$. Now, the maximal chains of the tuples $t'_0 = (\Delta'; \tau'; \tau)$ and $t'_1 = (\Delta'; \sigma; \sigma')$ are either maximal in t'' and the length of the maximal chain is the same, or they are not and its maximal chains are smaller. Hence $\|t'_0\| \leq \|t\|$ and $\|t'_1\| \leq \|t\|$.

□

Given these intermediate results, we now prove that, modulo similarity, the number of reachable tuples is finite for all inputs.

Lemma 5.33. *For any $t \in \mathcal{J}$, $Reach(t)_{/\simeq}$ is finite.*

Proof. Let $t \triangleq (\Delta, \tau, \sigma)$ and for the sake of getting a contradiction, suppose that $Reach(t)_{/\simeq}$ is infinite. Then there is an infinite sequence of non-similar tuples $t_0, t_1, t_2, \ldots$ reachable from t where $t_i = (\Delta, \tau_i, \sigma_i)$. By Lemma 5.28 all types occurring in tuples reachable from t are subexpressions of the types occurring in t according to Definition 5.26. By Lemma 5.32 we also know that all tuples have maximal chains with a length smaller than the length of a maximal chain of t.

So, using the similarity relation, we can define an infinite sequence $t'_0, t'_1, t'_2, \ldots$ of non-similar tuples where $t'_i \simeq t_i$ and whose environments only contain the initial variables in Δ and the variables necessary to support the free variables of τ_i and σ_i. However, the number of subexpressions of τ and σ is finite up to renaming of type variables, and the number of chains in the typing environment in any t'_i is bounded by the number of free variables in τ_i and σ_i and, by Lemma 5.32, their length is bounded by the maximal chain of t. In these conditions it is obvious that the number of tuples that can be defined from Δ, a limited number of type variables, and a limited number of type expressions is finite up to renaming of type variables. Since each one of the tuples t'_i is similar to a different tuple in the set of all possible tuples (otherwise they would be similar by transitivity) we reach a contradiction. Hence, $Reach(t)_{/\simeq}$ must be finite. □

In this finite search space, and by observing the algorithm, we can immediately conclude that it must terminate, since it does not repeat the computation for two similar tuples. This is formally stated in the following theorem:

Theorem 5.34 (Termination). *For all sets of tuples $A \in \mathcal{P}(\mathcal{J})$, and tuples $t \in \mathcal{J}$, $Subtyping(A, t)$ terminates.*

Proof. We prove that the $\mathit{Subtyping}(A,t)$ terminates on all inputs by induction on a customly defined measure. We assign to each subsidiary call $\mathit{Subtyping}(A',t')$ of the subtyping algorithm a measure given by

$$|\mathit{Subtyping}(A',t')| \triangleq \#\mathit{Reach}(t)_{/\simeq} - \#(A'_{/\simeq})$$

for all A' and t' such that $t \gg t'$. The measure gives the number of equivalence classes in $\mathit{Reach}(t)_{/\simeq}$ that were not yet visited by the algorithm.

When $|\mathit{Subtyping}(A',t')| = 0$ then there is $t'' \in A'$ with $t'' \simeq t'$. t'' acts as a representative for that equivalence class modulo similarity of t'. So, the algorithm terminates returning A'.

For the inductive case we have $|\mathit{Subtyping}(A',t')| > 0$. There are two possible cases: either $t' \in^{\simeq} A$ and the algorithm terminates with A', or $t' \notin^{\simeq} A$. In this case, we can see that $|\mathit{Subtyping}(A',t')| > |\mathit{Subtyping}(A' \cup \{t'\},u)|$ for any u such that $t \gg u$. Therefore, by the induction hypothesis all recursive calls of the subtyping algorithm terminate, and so the initial call also terminates. □

It is important to remark that the finite reachability property of Lemma 5.33 holds both for kernel-Fun and $F_{\leq}^{\top}$. From the proof we can also see why the same result cannot be extended to $F_{\leq}$. Notice that by using the rule for quantified types in $F_{\leq}$,

$$\frac{\Delta \vdash \delta' \leq \delta \quad \Delta, X \leq \delta' \vdash \tau \leq \tau'}{\Delta \vdash \forall_{X \leq \delta}\, \tau \leq \forall_{X \leq \delta'}\, \tau'}$$

the variable chains containing variables from δ, X and continuing in τ are altered in the premises to start in δ'. This may cause larger variable chains to appear in reachable tuples and therefore cause a divergence of the algorithm.

Now, to prove that our algorithm is sound and complete, it is technically convenient to follow the approach of [48] and introduce a function *gfp* that characterises νS in a form both suitable for the correctness proofs and for establishing the correspondence between the algorithm and the extensional definition of the subtyping relation. Moreover, unlike the analogous notion in [48], instead of accumulating tuples, our *gfp* function works with $\simeq$-equivalence classes.

Definition 5.35 (Support). *$\mathit{support}(t)$ denotes the minimal set $G \in \mathcal{P}(\mathcal{J})$ such that $t \in S(G)$.*

Definition 5.36 (gfp). *Let gfp be the partial function $\mathcal{P}(\mathcal{J}) \times \mathcal{J} \to \mathcal{P}(\mathcal{J})$ defined by:*

```
$gfp(A,t) =$  if $t^* \subseteq A$ then $A$
              else if $support(t)$ is undefined then undefined
              else let $\{t_1, \ldots, t_n\} = support(t)$ in
                let $A_0 = A \cup t^*$ in
                let $A_1 = gfp(A_0, t_1)$ in
                ...
                let $A_n = gfp(A_{n-1}, t_n)$ in $A_n$.
```

Lemma 5.37. *For all $t \in \mathcal{J}$, if $t \in \nu S$ then $support(t) \subseteq \nu S$.*

Proof. Since the greatest fixed point of S is the union of all consistent sets (Tarski's theorem), if $t \in \nu S$ then there is a set $G \in \mathcal{P}(\mathcal{J})$ such that $t \in G$ and $G \subseteq S(G)$. By Definition 5.35, this implies that $support(t) \subseteq G$ which implies that $support(t) \subseteq \nu S$. □

The next lemma states that *gfp* correctly characterises the subtyping relation.

Lemma 5.38 (Correctness of *gfp*). *For all $t \in \mathcal{J}$, and $A \in \mathcal{P}(\mathcal{J})$,*

1. *if $gfp(\emptyset, t) = A$ then $t \in \nu S$.*

2. *if $gfp(\emptyset, t)$ is undefined then $t \notin \nu S$.*

Proof. To prove the first part of the lemma we prove an auxiliary result by induction on the definition of *gfp*. For all $A, A' \in \mathcal{P}(\mathcal{J})$, and $t \in \mathcal{J}$:

$$\text{If } A^* \subseteq A \text{ and } gfp(A,t) = A' \text{ then } A \subseteq A', A'^* \subseteq A', t^* \subseteq A', \text{ and } A' \subseteq S(A') \cup A$$

Remember that A^* is the closure of A under similarity and that $A^* \subseteq A$ means that A is closed under similarity.

The induction base is when $t^* \subseteq A$, we have that *gfp* yields $A' = A$ and all the conclusions result directly from the assumptions, and thus $A'^* \subseteq A'$, $t^* \subseteq A'$, and $A' \subseteq S(A') \cup A$.

In the inductive case we know that t^* and A are disjoint and $\{t_1, \ldots, t_n\} = support(t)$. Notice that A is closed under similarity and therefore $t^* \nsubseteq A$ implies that $t^* \cap A = \emptyset$. By applying the induction hypothesis to the recursive function calls:

$$\begin{array}{c} gfp(A_0, {t_1}^*) = A_1 \\ gfp(A_1, {t_2}^*) = A_2 \\ \ldots \\ gfp(A_{n-1}, {t_n}^*) = A_n \end{array}$$

we conclude in sequence that

$$\begin{gathered}A_0 \subseteq A_1,\ t_1{}^* \subseteq A_1 \text{ and } A_1 \subseteq S(A_1) \cup A_0,\\ A_1 \subseteq A_2,\ t_2{}^* \subseteq A_2 \text{ and } A_2 \subseteq S(A_2) \cup A_1,\\ \dots\\ A_{n-1} \subseteq A_n,\ t_n{}^* \subseteq A_n \text{ and } A_n \subseteq S(A_n) \cup A_{n-1}.\end{gathered}$$

and hence we obtain $A_n \subseteq S(A_n) \cup S(A_{n-1}) \cup \ldots \cup A_0$. By monotonicity of S, we have that $A_n \subseteq S(A_n) \cup A \cup \{t\}$. As we know that $t_1{}^* \cup t_2{}^* \cup \ldots \cup t_n{}^* \subseteq A_n$ and that $A_n{}^* \subseteq A_n$ then we conclude that for all $t' \in t^*$ there is a set of tuples $t'_i \in t_i{}^*\ {}^{i \in 1..n}$ such that $t^* \subseteq S(A_n)$ and hence $A_n \subseteq S(A_n) \cup A$. Now, with $A = \emptyset$ we conclude that $t^* \subseteq A'$, $t \in A'$, and $A' \subseteq S(A')$. By the coinduction principle this implies that $t \in \nu S$.

The second part of the lemma follows from another auxiliary property:

For all $A \in \mathcal{P}(\mathcal{J})$ and $t \in \mathcal{J}$, if $gfp(A, t^*)$ is undefined then t^* and νS are disjoint.

and to prove it we analyse the two possible reasons for $gfp(A, t^*)$ to be undefined:

Case: *$support(t)$* **is undefined**

For the sake of contradiction lets suppose that $t \in \nu S$. By Lemma 5.37, $support(t) \subseteq \nu S$ which is a contradiction. So indeed we have $t \notin \nu S$. Since, by Lemma 5.23, νS is closed under similarity we have that t^* and νS must be disjoint.

Case: $support(t) = \{t_1, \ldots, t_n\}$

and there is a least j such that $gfp(A_{j-1}, t_j)$ is undefined. By the induction hypothesis, we have $t_j{}^*$ and νS are disjoint. By Lemma 5.37, if $t \in \nu S$ then $support(t) \subseteq \nu S$ which is again a contradiction, since $support(t) \cap t_j{}^* \neq \emptyset$. □

Knowing that *gfp* correctly checks if any tuple belongs to the subtyping relation νS, we now prove that the results of *gfp* and *Subtyping* are equivalent. Since the algorithm terminates on all inputs (Theorem 5.34), we show as a consequence that the algorithm is sound and complete with relation to the subtyping relation.

Lemma 5.39 (Correctness of *Subtyping*)**.** *For all $t \in \mathcal{J}$, $A \in \mathcal{P}(\mathcal{J})$, and $A' \in \mathcal{P}(\mathcal{J})$,*

1. *$Subtyping(A, t) = A'$ if and only if $gfp(A^*, t) = A'^*$.*

2. *$Subtyping(A, t) = fail$ if and only if $gfp(A^*, t)$ is undefined.*

Proof. We divide the first case of the proof in the two directions of the equivalence. We prove the "only if" part of the lemma by induction on the recursive calls of *Subtyping* and in the cases of the algorithm.

Case: $t \in^{\simeq} A$

In this case we have that $A' = A$ and $t^* \subseteq A^*$. Hence $\mathit{gfp}(A^*, t) = A^*$ with $A'^* = A^*$.

Case: t **matches a case of** $\mathit{Subtyping}$

Then we know that $\mathit{support}(t)$ is defined and by inspection we conclude that all recursive calls of $\mathit{Subtyping}$ correspond to the sequence of calls in *gfp* for the tuples in the support of t. Note that $A' = A_n$. By applying the induction hypothesis to each recursive call of $\mathit{Subtyping}$ we conclude that each $\mathit{gfp}(A_{i-1}{}^*, t_i) = A_i{}^*$ with $A_0 = A^* \cup t^*$. Therefore $\mathit{gfp}(A^*, t) = A'^*$ with $A'^* = A_n{}^*$.

In the other direction, the "if" part of the lemma is proven by Lemma 5.34 and the second case of the lemma. As $\mathit{Subtyping}$ terminates on all inputs and so, if $\mathit{gfp}(A^*, t) = A'$, which (by 2) implies that there should be a A'' such that $\mathit{Subtyping}(A^*, t) = A''$ and by 1, $A'' = A'$.

The second part of the lemma (2), is again proven in two steps. The "only if" implication is proven by induction on the recursive calls of the $\mathit{Subtyping}$ algorithm and in the two possible cases for failure of $\mathit{Subtyping}(A, t)$.

Case: t **does not match any expected type shape**

If $\mathit{Subtyping}$ fails because t does not match any of the cases of the algorithm then $\mathit{support}(t)$ is undefined for that tuple. Hence $\mathit{gfp}(A^*, t)$ is also undefined.

Case: **A recursive call fails**

If on the other hand, a recursive call of $\mathit{Subtyping}$ fails, by induction hypothesis we know that the corresponding recursive call to *gfp* is undefined. Therefore $\mathit{gfp}(A^*, t)$ is undefined.

To complete the proof we show the other direction of the lemma. If $\mathit{gfp}(A^*, t)$ is undefined then $\mathit{Subtyping}(A, t) = \mathit{fail}$. If $\mathit{gfp}(A^*, t)$ is undefined then there is no A' such that $\mathit{gfp}(A^*, t) = A'$ then (by part 1) there is no A' such that $\mathit{Subtyping}(A, t) = A'$. Since $\mathit{Subtyping}$ always terminates, we must have $\mathit{Subtyping}(A, t) = \mathit{fail}$. □

We use *gfp* as an intermediate definition. By Lemma 5.38 we prove that *gfp* corresponds to our subtyping relation and in Lemma 5.39 we prove that its values correspond to the results of our algorithm with the same arguments. By transitivity we have the correctness of our subtyping algorithm as stated in the following theorem:

Theorem 5.40 (Correctness of $\mathit{Subtyping}$)**.** *For all $\Delta \in \mathcal{D}$, and $\tau, \sigma \in \mathcal{T}$,*

1. $\mathit{Subtyping}(\emptyset, (\Delta, \tau, \sigma)) = A$ if and only if $\Delta \vdash \tau \leq \sigma$

2. $\mathit{Subtyping}(\emptyset, (\Delta, \tau, \sigma)) = \mathit{fail}$ if and only if $\Delta \nvdash \tau \leq \sigma$

Proof. Both cases of the theorem follow immediately from Lemmas 5.38 and 5.39. For the first case we have that $\mathit{Subtyping}(\emptyset, (\Delta, \tau, \sigma)) = A$ if and only if $\mathit{gfp}(\emptyset, (\Delta, \tau, \sigma)) = A^*$, Lemma 5.39, and therefore, $(\Delta, \tau, \sigma) \in \nu S$, Lemma 5.38, which means that $\Delta \vdash \tau \leq \sigma$, Definition 5.10.

For the second, we know that $\mathit{Subtyping}(\emptyset, (\Delta, \tau, \sigma)) = \mathit{fail}$ if and only if $\mathit{gfp}(\emptyset, (\Delta, \tau, \sigma)) = \mathit{fail}$, Lemma 5.39, and therefore, $(\Delta, \tau, \sigma) \notin \nu S$, Lemma 5.38, which means that $\Delta \not\vdash \tau \leq \sigma$, Definition 5.10. □

This concludes our presentation of the subtyping algorithm for the relation of Definition 5.10. The subtyping relation is defined coinductively from a set of structural comparison rules to compare the natural interpretation of recursive types as infinite regular trees. This algorithm can be applied to any subtyping relation for second-order equi-recursive types where we can define a similarity relation such that the set of reachable tuples is finite modulo similarity.

The results presented in this section are used in the next chapter to present a new type system for our component language with subtyping, recursive types and bounded polymorphism.

5.3 Remarks

The work presented in this chapter evolved over a couple of years in parallel with the development of the type system for our component language. Early developments were done in the context of the type system for the component language. By then, the structure of the component types seem to lead to a decidable subtyping relation even with a $F_{\leq}$ subtyping discipline for polymorphic component types. That revealed to be untrue as we manage to encode the divergent cases of $F_{\leq}$ in our component language. The good side of this setback was that our syntactic impositions on type structure for components were not, after all, limiting its expressiveness.

As a result of studying coinductive subtyping definitions for first-order recursive types, and also the complex approach of Colazzo and Ghelli to second-order types with equi-recursive types [30, 31], we devised a much simpler approach for solving the problem for the $F_{\leq}^{\top}$ subtyping discipline, that also applies to the kernel-Fun style.

We then tried to streamline our results and isolate the initial results outside the context of the component language, which we managed to do in [87]. It focuses on a small class-based language with equi-recursive types and bounded quantification.

In that paper we also presented a composition mechanism for classes inspired in our component language that avoids the major type related problems of inheritance-based languages. This may lead to future results on the area of object-oriented languages.

The process of writing the final version of [87] and later this chapter lead to a cleaner version of the definition of subtyping relation and algorithm, this time for the pure kernel-Fun language.

Notice that the results obtained in this chapter depend on Lemma 5.33 and hence apply also to other subtyping disciplines that have a finite search-space modulo the similarity relation. It is clear that in the case of $F_{\leq}$, Lemma 5.32 does not hold, and this pinpoints the source of $F_{\leq}$

undecidability found by Pierce and Ghelli. If we analyse the algorithm in terms of variable chains we see that the chains of a variable may be changed in the course of a derivation which may cause for an increase on the length of the maximal chain of a judgement. However, this property applies to subtyping disciplines that fit well with the object-oriented style. One of this proposals is $F_{\leq}^{\top}$, which contra-variantly relates the bounds of quantified types but leaves the variables unbounded when relating quantified expressions. In this case the variables on one side only relate by reflexivity with the variables on the other side of the relation. Another variant comes from noticing that in usual class-based languages, object types are usually extended by adding more methods to it rather than refining their signatures. A subtyping relation that relates the bounds of type parameters in a width-only subtyping discipline, thus not allowing for the variable chains to change, again fits the necessary conditions to define a correct subtyping algorithm.

5.4 Related Work

Our work relates, by extension to second-order systems, with the work on subtyping first-order type systems with recursive types of Amadio and Cardelli [6], Brandt and Henglein [19], and Gapeyev et al. [48, 78].

On second-order systems with recursive types, our work closely relates with two other approaches, the one by Colazzo and Ghelli [30, 31] and another by Alan Jeffrey [60]. The pioneer work of Colazzo and Ghelli presents an approach to the problem which is a specific coinductive interpretation of an inductive rule set (similar to the one in Figure 5.1). They extend the initial algorithm of [6] which tracks the pairs of types which are found by the algorithm along the structure of types, and show that the simple extension of this algorithm to second-order systems is not correct due to the introduction of new variable declarations in quantified types. They prove that, in their derivation system, the third time a pair is visited effectively corresponds to a cycle in the derivation and therefore, by coinduction, correspond to a successful derivation. Unlike them, we explicitly rename variables when analysing quantified types and therefore avoid positively matching variables that are different. We keep the variable's context in the judgement and by defining a similarity relation on judgements we take the whole context into account, and abstract from variable names.

Colazzo presents two examples where simpler algorithms either diverge or yield wrong results. The first case diverges when regular α-renaming of variables on quantifiers is used and when the algorithm stops when a pair of types is revisited. Our algorithm stops the first time the judgement repeats itself (modulo similarity). We present here an abbreviation of the derivation produced by an ML implementation of our algorithm [82], which shows the intermediate and terminal cases that match modulo similarity. Given the following types:

$$U = \mu w_1.\forall w_2.\mu x.\forall t.(\bot \times \mu z.\forall w_3.\mu w_4.\forall w_5.(((\bot \times t) \times x) \times z))$$
$$T = \mu y.\forall u.\mu k.\forall w_6.(((u \times \top) \times k) \times y)$$

The derivation is as follows:

1. $\vdash U \leq T$

$\dots$

4. $v_0 \leq \top \vdash \mu x.\forall t.(\bot \times \mu z.\forall w_3.\mu w_4.\forall w_5.(((\bot \times t) \times x) \times z)) \leq \mu k.\forall w_6.(((v_0 \times \top) \times k) \times T)$

$\dots$

10. $v_1 \leq \top, \dots \vdash \mu z.\forall w_3.\mu w_4.\forall w_5.(((\bot \times v_1) \times \mu x.\forall t.(\bot \times \mu z.\forall w_3.\mu w_4.\forall w_5.(((\bot \times t) \times x) \times z))) \times z) \leq T$

$\dots$

23. $\dots, v_2 \leq \top, \dots \vdash \mu x.\forall t.(\bot \times \mu z.\forall w_3.\mu w_4.\forall w_5.(((\bot \times t) \times x) \times z)) \leq \mu k.\forall w_6.(((v_2 \times \top) \times k) \times T)$
24. $\dots, v_1 \leq \top, \dots \vdash \mu z.\forall w_3.\mu w_4.\forall w_5.(((\bot \times v_1) \times \mu x.\forall t.(\bot \times \mu z.\forall w_3.\mu w_4.\forall w_5.(((\bot \times t) \times x) \times z))) \times z) \leq T$

Notice that the subtyping of quantified types introduces fresh variables represented by v_0, v_1, etc, and although v_0 and v_2 represent different variables, the judgements in steps 4 and 23 match and terminate two derivation branches. Apart from the extra variables in the environment steps 10 and 24 are equal. Without the similarity relation on judgements, the derivation that would follow step 23 would diverge.

In the second case, an algorithm based on similarity is shown to be unsound. Hereafter, we show a correct derivation by our algorithm. Consider the following types:

$$U = \mu z.\forall t.\mu x.(x \times (t \times z))$$
$$T = \forall u.\mu y.((\top \times (u \times \forall v.y)) \times \top)$$

Then, the derivation of the subtyping judgement is as follows:

$U = \mu z.\forall t.\mu x.(x \times (t \times z))$
$T = \forall u.\mu y.((\top \times (u \times \forall v.y)) \times \top)$
$\vdash U \leq T$

1. $\vdash \forall t.\mu x.(x \times (t \times U)) \leq \forall u.\mu y.((\top \times (u \times \forall v.y)) \times \top)$
2. $v_0 \leq \top \vdash \mu x.(x \times (v_0 \times U)) \leq \mu y.((\top \times (v_0 \times \forall v.y)) \times \top)$
3. $v_0 \leq \top \vdash (\mu x.(x \times (v_0 \times U)) \times (v_0 \times U)) \leq \mu y.((\top \times (v_0 \times \forall v.y)) \times \top)$
4. $v_0 \leq \top \vdash (\mu x.(x \times (v_0 \times U)) \times (v_0 \times U)) \leq ((\top \times (v_0 \times \forall v.\mu y.((\top \times (v_0 \times \forall v.y)) \times \top))) \times \top)$

$\dots$

15. $v_1 \leq \top, v_0 \leq \top \vdash (\mu x.(x \times (v_1 \times U)) \times (v_1 \times U)) \leq ((\top \times (v_0 \times \forall v.\mu y.((\top \times (v_0 \times \forall v.y)) \times \top))) \times \top)$

$\dots$

20. $v_1 \leq \top, v_0 \leq \top \vdash v_1 \leq v_0$
21. $v_1 \leq \top, v_0 \leq \top \vdash \top \leq v_0$

$fail$

Notice that, in steps 4. and 15., although we may consider that the types involved are similar, the usage of v_0 and v_1 differ in the two judgements, hence the two judgements are not in the similarity relation. What separates our approach from the one of Colazzo and Ghelli is the fact that we do not keep the visited pairs "in" the judgement, with the consequent difficulties when maintaining and comparing type variables, but instead compare the types together with the contexts of the type variables (modulo syntactic equivalence).

Alan Jeffrey presents a partially correct algorithm for the $F_{\leq}$ subtyping relation with equi-recursive types [60] which is shown to be correct in the case of kernel-Fun. The algorithm is based on an encoding of types in labelled transition systems and subtyping is mapped into a simulation up to a special substitution definition. It decides, given a pair of types and an environment, if such a simulation exists. The algorithm proceeds in a breadth-first manner by gradually building a simulation between the two types modulo silent transitions, which correspond to the unfolding of recursive types.

On the other hand, we present a simple syntax-based algorithm that uniformly extends standard approaches for first-class systems and uses a simple similarity relation on judgements whose implementation is straightforward.

An efficient algorithm to unify two recursive second-order types was proposed by Gauthier and Pottier in [50]. It relies on an encoding of second-order type expressions into first-order trees, and on the application of standard first-order unification algorithms for infinite trees. We have no perspective on how this may be adapted to the subtyping problem.

Chapter 6

Polymorphic Components and Recursive Types

The component-based programming model that we have presented in the preceding chapters has been defined in a succession of core programming languages where dynamic composition and reconfiguration of objects is combined with the free manipulation of first-class components and configuration scripts.

In this chapter, we bring our model to a more finished design by combining the language mechanisms that express dynamic composition and reconfiguration of objects with standard notions of polymorphism and type recursion. The notion of subtyping (subsumption of values) defines a notion of inclusion polymorphism allowing the safe substitution of values with different sorts in statically typed contexts. In this case we define a subtyping relation on component types based on the declared required and provided services. Bounded quantification of type variables, combining parametric polymorphism with subtyping, forms a polymorphic mechanism specially suitable for object-oriented languages. Since the subsumption relation on objects usually leads to partial views of their interfaces, bounded quantification, allows for the construction of software pieces that can (partially) interact with abstractly typed values in a type safe manner. Bounded quantification has been recognised as such even in the most pragmatic and widely known main-stream languages like C♯ [61] and Java [59]. Finally, type recursion turns out to be essential in any object-oriented language to represent operations and relations between objects of the same sort.

In this chapter we define the programming language $\lambda_{\chi}^{\leq}$ by extension of the programming language λ_{ρ}, presented in Chapter 4, with subtyping, parametric polymorphism, and recursive types. We define a structural subtyping relation on the types of $\lambda_{\chi}^{\leq}$ and show how the general results obtained in Chapter 5 about the subtyping of second-order recursive types can be used to support subtyping, polymorphism, and type recursion in a more complex setting such as our component-based programming language. We show that the typing problem of our component-based language is decidable by presenting a typing algorithm and a subtyping algorithm. We prove that our typing algorithm is sound and complete by showing that it enjoys the property of minimal-typing.

This chapter is therefore organised as follows: In section 6.1 we discuss the design of the subtyping relation on component types. We then defined the language $\lambda_{\chi}^{\leq}$, in Section 6.2, by extending the programming language λ_{ρ} with the basic mechanisms just described and proving type safety for $\lambda_{\chi}^{\leq}$. Section 6.3 defines correct typing and subtyping algorithms and shows that the typing problem is decidable.

We then present a draft idea of a rich subtyping relation for configurators and discuss some of the ideas introduced. We discuss, in Section 6.6, the implementation of a prototype compiler of componentJ, a programming language inspired in the concepts of $\lambda_{\chi}^{\leq}$ and targeting the Java platform. In closing, we discuss some of the issues involving the integration of the different parts of the programming model.

6.1 Subtyping Based on Services

Until this point, we have not yet defined, for the component language, any equivalence relation on types. Hence, types are only related by an implicit equivalence relation which has been kept at the syntactic level. By defining a subtyping relation on types, based on the semantics of the values rather than the syntax of types [69, 27], we define a more flexible typing relation by extending the set of values accepted in a number of situations.

It is common for object-oriented language designers to adopt name-based subtyping relations which are naturally associated to the inheritance mechanism, even in research languages, e.g. FJ [59]. However, the use of late binding, dynamic loading, and mobile code, which motivated our work on structural subtyping in Chapter 5, suggests that some kind of structural subtyping is essential to ensure low-level compatibility between separately developed components in a wide scale environment.

Hence, we define on top of a standard subtyping relation for the base language (λ_R) a structural subtyping relation for component and object types. The principle of safe substitution on components can be intuitively understood in terms of the services that components require and provide. A component value must provide at least the services that are expected in the utilisation context (statically typed) and these provided services may be more specific (of a subtype) with relation to the expected type; furthermore, components may require less services from its context than expected, and the specification of these required services may be less specific (of a supertype).

The subtyping relation on component types extends uniformly to object types but referring only the provided services: an object can be used in a given context only if it provides the expected services; as in the component type relation the provided services of the object may be more specific than expected and the object may provide more services which are not used.

Configurator types can also be related so that the effect they cause is subsumed. A configurator type defines a precondition to the application of its values, which give a partial view of the elements it interacts with, and a postcondition, which gives the relative view of its effect. Thus, a subtyping relation on configurator types may be defined by defining subtypes with more restricted views on required resources, i.e. less available elements, more unsatisfied resources, and introducing more available elements and less unsatisfied resources. We do not integrate the subtyping of configurator types in the design of $\lambda_\chi^\leq$ but we explain the intuition that supports it in section 6.4.

We illustrate, by a simple example, the subtyping relation on component types.

Example 6.1. In this example, we use a factory function to create components out of given arguments. We illustrate subsumption on component values and show that it preserves the structure of the resulting structures. Consider the following type abbreviations for two interface types:

```
I = {m1:int → int}
J = {m1:int → int, m2:int → int}
```

and a component type:

```
T = {q:J}⇒{p:I}
```

Observe that in the usual sense we have that J is a subtype of I. Consider the following expression:

```
let f = fun x:T → compose(provides p:I;
                          m[m1 = fun x:int → x];
                          c[x:T];
                          plug m into c.q;
                          plug c.p into p) in
let D = compose(provides p:J;
                m[m1 = fun x:int → 2*x,
                  m2 = fun x:int → 2*x];
                plug m into p) in
let o = new f(D) in ...
```

Notice that in the definition of function f, the formal parameter x is used to introduce an internal element in the component resulting from calling the function.

The component type T that types x, allows for the correct typing of the plug operations between the ports of the internal component x and its context. The connection of method block m to port c.q and the connection of port c.p to port p.

The argument passed in the call to function f, which is component D defined above, is typed {}⇒{p:I} (which is a subtype of T). The component type indicates that it implements the service specified in port p without need for any external reference.

Notice that component D does not require any port as it provides a port whose type is a subtype of the demanded in the static type of x. Notice that the soundness of the resulting structure remains intact, i.e. no dependencies are left unsatisfied. By instantiating the internal component x with the component value D there is a plug operation in the composition in function f composition that produces no practical effect (the one that connects method block o to port c.q).

Now that we have described a subtyping relation we define it in a core component-based programming language with bounded quantification and recursive types.

6.2 $\lambda_{\chi}^{\leq}$ — A Polymorphic Component Language with Recursive Types

In this section, we define $\lambda_{\chi}^{\leq}$ by using the results of Chapter 5 and extending the component-based language λ_{ρ}, defined in Chapter 4. The new features of $\lambda_{\chi}^{\leq}$ are subtyping, bounded parametric polymorphism, and equi-recursive types.

Types

We first introduce the type language for $\lambda_{\chi}^{\leq}$ by building on $\mathcal{T}_{\chi}$ and adding the necessary expressions to express polymorphic and recursive types. Let $\mathcal{Z}$ be a denumerable set of type variables.

Definition 6.2 (Types). *The types $\mathcal{T}_{\chi}^{\leq}$ of $\lambda_{\chi}^{\leq}$ are defined by the abstract syntax in Figure 6.1.*

The occurrence of the type variable X in the type expression $\forall_{X \leq \delta}\ \tau$ is binding in τ. This type expression also states that all possible instantiations (types) for the type variable X should be subtypes of type δ. The type expression $\top$ is helpful for introducing unbound type variables in the language. Recursive types, of the form $\mu X.\tau$, already used in Chapter 5 are also added to the type language.

We now extend our language $\lambda_{\chi}^{\leq}$ by adding the standard terms for type abstraction and type application to the expressions of λ_{ρ}, as follows:

$$
\begin{array}{lcll}
\tau & ::= & & \text{types} \\
 & | & X & \text{type variable} \\
 & | & \top & \text{top} \\
 & | & \forall_{X \leq \tau}\ \tau & \text{bounded quantification} \\
 & | & \mu X.\tau & \text{recursive types} \\
 & | & \tau \rightarrow \tau & \text{function type} \\
 & | & \{\!|\ell_i : \tau_i\ ^{i \in 1..n}|\!\} & \text{record type} \\
 & | & \{\ell_i : \tau_i\ ^{i \in 1..n}\} & \text{interface type} \\
 & | & \tau \Rightarrow \tau & \text{component type} \\
 & | & \{r_i\ ^{i \in 1..n}\} \Longrightarrow \{r_i\ ^{i \in 1..m}\} & \text{configurator type} \\
 & & & \\
r & ::= & & \text{resources} \\
 & | & \pi \circ \tau & \text{unsatisfied resource} \\
 & | & \pi \bullet \tau & \text{available resource} \\
 & | & \pi \triangleright \tau & \text{provided resource} \\
 & | & \pi \triangleleft \tau & \text{required resource}
\end{array}
$$

Figure 6.1: Abstract syntax of $\lambda_{\chi}^{\leq}$ types.

e	$::=$		terms
	$\mid$	x	variable
	$\mid$	$\lambda x : \tau . e$	abstraction
	$\mid$	$e(e)$	application
	$\mid$	$\Lambda X \leq \tau . e$	type abstraction
	$\mid$	$e\langle\tau\rangle$	type instantiation
	$\mid$	$\{\ell_i = e_i \ ^{i \in 1..n}\}$	record
	$\mid$	$e.\ell$	selection
	$\mid$	$e.\ell := e$	assignment
	$\mid$	ℓ	port label
	$\mid$	$\mathsf{compose}\ \ e$	component creation
	$\mid$	$\mathsf{new}\ e\ \mathsf{with}\ \ell_j := e_j \ ^{j \in 1..m}$	instantiation
	$\mid$	$\mathsf{reconfig}\ x = e[e]\ \mathsf{with}\ \ell_i := e_i \ ^{i \in 1..n}\ \mathsf{in}\ e\ \mathsf{else}\ e$	reconfiguration
	$\mid$	$\mathsf{requires}\ \ell : \tau$	required port
	$\mid$	$\mathsf{provides}\ \ell : \tau$	provided port
	$\mid$	$x[e : \tau]$	component introduction
	$\mid$	$x_I[\ell_i : \tau_i = \lambda x : \tau . e_i \ ^{i \in 1..n}]$	method block
	$\mid$	$\mathsf{plug}\ \pi : \tau\ \mathsf{into}\ \pi : \tau$	plug
	$\mid$	$e; e$	configurator composition
	$\mid$	ι	location value
	$\mid$	nil	null value
	$\mid$	$(e, e, e)_\Gamma$	object
	$\mid$	$\mathsf{comp}(e)$	component
	$\mid$	$\mathsf{conf}(\tau, e)$	configurator
π	$::=$	$\ell \mid x \mid x.\ell$	port name

Figure 6.2: Abstract syntax of $\lambda^{\leq}_{\chi}$.

$$
\begin{array}{lcll}
v & ::= & & \text{values} \\
 & | & \lambda x : \tau . e & \text{abstraction} \\
 & | & \Lambda X \leq \tau . e & \text{type abstraction} \\
 & | & \{\ell_i = \imath_i \ ^{i \in 1..n}\} & \text{record} \\
 & | & \imath & \text{location value} \\
 & | & \mathsf{nil} & \text{null value} \\
 & | & (r, r, r)_{\Gamma} & \text{object} \\
 & | & \mathsf{conf}(\tau, c) & \text{configurator} \\
 & | & \mathsf{comp}(c) & \text{component} \\
 & & & \\
r & ::= & \{\ell_i = \imath_i \ ^{i \in 1..n}\} & \text{record} \\
 & & & \\
c & ::= & & \text{ground composition operations} \\
 & | & \mathsf{requires}\ \ell : \tau & \text{required port} \\
 & | & \mathsf{provides}\ \ell : \tau & \text{provided port} \\
 & | & x[v : \tau] & \text{internal component} \\
 & | & x_I[\ell_i : \tau_i = \lambda x_i . e_i \ ^{i \in 1..n}] & \text{method block} \\
 & | & \mathsf{plug}\ \pi : \tau\ \mathsf{into}\ \pi : \tau & \text{plug} \\
 & | & c ; c & \text{composition} \\
 & & & \\
\pi & ::= & \ell \mid x \mid x.\ell & \text{port name}
\end{array}
$$

Figure 6.3: Abstract syntax of $\lambda^{\leq}_{\chi}$ values.

Definition 6.3 (Terms). *The language $\lambda^{\leq}_{\chi}$ is defined by the abstract syntax in Figure 6.2.*

As expected, the set of $\lambda^{\leq}_{\chi}$ values is obtained by adding type abstractions to the set of expressions that may result from evaluating expressions.

Definition 6.4 (Values). *The set of values $\mathcal{U}^{\leq}_{\chi} \subseteq \lambda^{\leq}_{\chi}$ is defined by the abstract syntax in Figure 6.3.*

Notice that type abstraction and application is added to the language λ_ρ in a compositional way, e.g. type abstraction can be uniformly applied to any expression.

We use the usual definition of capture avoiding substitution of types in type expressions and terms. Notice that the substitution of variables in terms extends uniformly to the new expression for type abstractions. Consider that for a substitution θ that we have

$$(\Lambda X \leq \tau . e)\theta \triangleq \Lambda X \leq \tau . (e\theta)$$

We now define the notion of free type variables in a type expression as follows:

Definition 6.5. *We define the set $FTV(\tau)$ of free variables of type expression τ as follows:*

$$\begin{aligned}
FTV(X) &\triangleq X \\
FTV(\top) &\triangleq \emptyset \\
FTV(\forall_{X\leq\tau}\ \sigma) &\triangleq FTV(\tau) \cup FTV(\sigma)\backslash\{X\} \\
FTV(\mu X.\tau) &\triangleq FTV(\tau)\backslash\{X\} \\
FTV(\tau \to \tau) &\triangleq FTV(\tau) \cup FTV(\sigma) \\
FTV(\{\!|\ell_i : \tau_i\ ^{i\in 1..n}|\!\}) &\triangleq \textstyle\bigcup_{i\in 1..n} FTV(\tau_i) \\
FTV(\{\ell_i : \tau_i\ ^{i\in 1..n}\}) &\triangleq \textstyle\bigcup_{i\in 1..n} FTV(\tau_i) \\
FTV(\tau \Rightarrow \sigma) &\triangleq FTV(\tau) \cup FTV(\sigma) \\
FTV(\{\ell * \tau^{i\in 1..n}\} \Longrightarrow \{\ell' * \tau'^{i\in 1..m}\}) &\triangleq \textstyle\bigcup_{i\in 1..n} FTV(\tau_i) \cup \bigcup_{i\in 1..m} FTV(\tau'_i) \\
&\textit{where } * = \bullet, \circ, \lhd, \rhd.
\end{aligned}$$

The notion of substitutions of type variables in type expressions and in terms is defined as follows:

Definition 6.6 (Type Substitution)**.** *A substitution (Θ) is a finite mapping from type variables to type expressions.*

We denote by $[X \leftarrow \tau]$ the singleton substitution that maps X to τ. We write $\mathrm{Dom}(\Theta)$ to denote the domain of the substitution Θ and define the codomain of a substitution Θ by $\mathrm{Img}(\Theta) \triangleq \bigcup\{FTV(\Theta(X)) \mid X \in \mathrm{Dom}(\Theta)\}$. We write $\Theta\downharpoonleft_X$ to restrict the domain of the substitution Θ by eliminating the substitution of X.

Definition 6.7 (Application of type substitution in type expressions)**.** *We define the application of a substitution Θ to an expression τ, written $\tau\Theta$, as follows:*

$$\begin{aligned}
X\Theta &\triangleq \Theta(X) \\
X\Theta &\triangleq X \textit{ if } X \notin \mathrm{Dom}(\Theta) \\
\top\Theta &\triangleq \top \\
(\forall_{X\leq\tau}\ \sigma)\Theta &\triangleq \forall_{X\leq\tau\Theta'}\ (\sigma\Theta') \textit{ where } \Theta' = \Theta\downharpoonleft_X \\
(\mu X.\sigma)\Theta &\triangleq \mu X.(\sigma\Theta') \textit{ where } \Theta' = \Theta\downharpoonleft_X \\
(\tau \to \sigma)\Theta &\triangleq (\tau\Theta) \to (\sigma\Theta) \\
\{\!|\ell_i : \tau_i\ ^{i\in 1..n}|\!\}\Theta &\triangleq \{\!|\ell_i : \tau_i\Theta\ ^{i\in 1..n}|\!\} \\
\{\ell_i : \tau_i\ ^{i\in 1..n}\}\Theta &\triangleq \{\ell_i : \tau_i\Theta\ ^{i\in 1..n}\} \\
(\tau \Rightarrow \sigma)\Theta &\triangleq (\tau\Theta) \Rightarrow (\sigma\Theta) \\
(\{r_i\ ^{i\in 1..n}\} \Longrightarrow \{r_i\ ^{i\in 1..m}\})\Theta &\triangleq \{r_i\Theta\ ^{i\in 1..n}\} \Longrightarrow \{r_i\Theta\ ^{i\in 1..m}\}
\end{aligned}$$

$$\textit{where, for each } * = \bullet, \circ, \rhd, \lhd, \textit{ and } r = \ell * \tau, \textit{ we have } r\Theta \triangleq \ell * (\tau\Theta).$$

$$\text{(Eval Type Application)}\quad \frac{e;S \downarrow \Lambda X \leq \tau'.e';S' \quad e'[X \leftarrow \tau];S' \downarrow v;S''}{e\langle\tau\rangle;S \downarrow v;S''}$$

Figure 6.4: Evaluation rule for $\lambda^{\leq}_{\chi}$.

Definition 6.8 (Application of type substitution in terms)**.** *We define the application of a substitution Θ to an expression e, written $e\Theta$, as follows:*

$$\begin{aligned}
x\Theta &\triangleq x\\
(\lambda x:\tau.e)\Theta &\triangleq \lambda x:(\tau\Theta).e\Theta\\
(\Lambda X \leq \tau.e)\Theta &\triangleq \Lambda X \leq \tau.(e\Theta')\ \textit{where}\ \Theta' = \Theta' \downharpoonleft_X\\
(e_1(e_2))\Theta &\triangleq e_1\Theta(e_2\Theta)\\
\{\ell_i = e_i\ ^{i\in 1..n}\}\Theta &\triangleq \{\ell_i = e_i\Theta\ ^{i\in 1..n}\}\\
(e.\ell)\Theta &\triangleq (e\Theta).\ell\\
(e_1.\ell = e_2)\Theta &\triangleq (e_1\Theta).\ell = (e_2\Theta)\\
(\mathsf{compose}\ e)\Theta &\triangleq \mathsf{compose}\ (e\Theta)\\
(\mathsf{new}\ e\ \mathsf{with}\ \ell_j := e_j\ ^{j\in 1..m})\Theta &\triangleq \mathsf{new}\ (e\Theta)\ \mathsf{with}\ \ell_j := (e_j\Theta)\ ^{j\in 1..m}\\
(\mathsf{requires}\ \ell:\tau)\Theta &\triangleq \mathsf{requires}\ \ell:(\tau\Theta)\\
(\mathsf{provides}\ \ell:\tau)\Theta &\triangleq \mathsf{provides}\ \ell:(\tau\Theta)\\
(x[e:\tau])\Theta &\triangleq x[(e\Theta):(\tau\Theta)]\\
(x[\ell_i:\tau_i = \lambda x_i:\tau'_i.e_i\ ^{i\in 1..n}])\Theta &\triangleq x[\ell_i:(\tau_i\Theta) = \lambda x_i:(\tau'_i\Theta).(e_i\Theta)\ ^{i\in 1..n}]\\
(\mathsf{plug}\ \pi_1:\tau_1\ \mathsf{into}\ \pi_2:\tau_2)\Theta &\triangleq \mathsf{plug}\ \pi_1:(\tau_1\Theta)\ \mathsf{into}\ \pi_2:(\tau_2\Theta)\\
(e_1;e_2)\Theta &\triangleq (e_1\Theta);(e_2\Theta)\\
\imath\Theta &\triangleq \imath\\
\mathsf{nil}\Theta &\triangleq \mathsf{nil}\\
\mathsf{conf}(\tau,e)\Theta &\triangleq \mathsf{conf}(\tau\Theta,e\Theta)\\
\mathsf{comp}(e)\Theta &\triangleq \mathsf{comp}(e\Theta)
\end{aligned}$$

We are now ready to define the operational semantics of $\lambda^{\leq}_{\chi}$.

Operational Semantics

Definition 6.9 (Evaluation)**.** *Let $e \in \lambda^{\leq}_{\chi}$ and a heap S such that $(e;S)$ is a valid configuration. The evaluation relation of an expression e to a value v, written $e;S \downarrow v;S'$ is defined inductively by the rules in Figures 2.8, 4.4, 4.5, 4.6, 4.7, and 6.4.*

The operational semantics of the language $\lambda^{\leq}_{\chi}$ is defined by extending the semantics of λ_ρ with a rule to evaluate type applications. Notice that the evaluation of type abstractions is already covered by Rule (Eval Value) and that the result of a type application is obtained by Rule (Eval Type Application). The evaluation proceeds by replacing the type parameter by the actual type argument.

$$
\begin{array}{lcll}
\Delta & ::= & & \text{typing environments} \\
& \mid & \phi & \text{empty environment} \\
& \mid & \Delta, x : \tau & \text{type assignment to a variable} \\
& \mid & \Delta, \imath : \tau & \text{type assignment to a location} \\
& \mid & \Delta, X \leq \tau & \text{bounded type variable declaration}
\end{array}
$$

Figure 6.5: Abstract syntax of typing environments in $\lambda_{\chi}^{\leq}$.

$$
\begin{array}{ccc}
\text{(Env Type Var)} & \text{(Type Top)} & \text{(Type TVar)} \\
\dfrac{\Delta \vdash \tau \;\mathsf{ok} \quad X \notin \mathrm{Dom}(\Delta)}{\Delta, X \leq \tau \vdash \diamond} & \dfrac{\Delta \vdash \diamond}{\Delta \vdash \top \;\mathsf{ok}} & \dfrac{\Delta \vdash \diamond \quad X \in \mathrm{Dom}(\Delta)}{\Delta \vdash X \;\mathsf{ok}}
\end{array}
$$

$$
\begin{array}{cc}
\text{(Type All)} & \text{(Type Rec)} \\
\dfrac{\Delta \vdash \delta \;\mathsf{ok} \quad \Delta, X \leq \delta \vdash \tau \;\mathsf{ok}}{\Delta \vdash \forall_{X \leq \delta}\, \tau \;\mathsf{ok}} & \dfrac{\Delta, X \leq \top \vdash \tau \;\mathsf{ok}}{\Delta \vdash \mu X.\tau \;\mathsf{ok}}
\end{array}
$$

Figure 6.6: Validation rules for typing environments in $\lambda_{\chi}^{\leq}$.

Type System

In order to define the type system we first need to extend the earlier definition of typing environments (Definition 2.17) with the declaration of bounded type variables. We also need to define a subtyping relation which we the use to define the typing relation of the language $\lambda_{\chi}^{\leq}$.

Definition 6.10 (Typing Environment). *For $x \in \mathcal{V}$, $X \in \mathcal{Z}$, $\imath \in \mathtt{Loc}$, and $\tau \in \mathcal{T}_{\chi}^{\leq}$ the set $\mathcal{D}^{\leq}$ of all typing environments is defined by the abstract syntax in Figure 6.5.*

Definition 6.11 (Valid typing environment). *A typing environment Δ is* valid *if the judgement $\Delta \vdash \diamond$ is derivable by the rules in Figures 2.10 and 6.6.*

We now define our subtyping relation for $\lambda_{\chi}^{\leq}$.

Subtyping

The subtyping relation between the types of $\lambda_{\chi}^{\leq}$ follows standard approaches to structural subtyping relations. The types are related according to their shape in ways that can be expressed by the Rules of Figure 6.7. Notice that these rules are not intended to define the subtyping relation but just to give an intuition about it. The actual subtyping relation we are interested in will be defined as the greatest fixed point of a generating function, following the approach of Chapter 5. Apart from the standard reflexivity, maximality of $\top$, the standard relation of

(Sub Equal)

$$\Delta \vdash \tau \leq \tau$$

(Sub Top)

$$\Delta \vdash \tau \leq \top$$

(Sub Trans)

$$\frac{\Delta \vdash \tau \leq \delta \qquad \Delta \vdash \delta \leq \sigma}{\Delta \vdash \tau \leq \sigma}$$

(Sub Fun)

$$\frac{\Delta \vdash \tau' \leq \tau \quad \Delta \vdash \sigma \leq \sigma'}{\Delta \vdash \tau \to \sigma \leq \tau' \to \sigma'}$$

(Sub Interface) $(n \leq m)$

$$\frac{\Delta \vdash \tau_i \leq \sigma_i \quad \forall i \in 1..n}{\Delta \vdash \{\ell_i : \tau_i{}^{\,i \in 1..m}\} \leq \{\ell_i : \sigma_i{}^{\,i \in 1..n}\}}$$

(Sub Component)

$$\frac{\Delta \vdash \tau' \leq \tau \quad \Delta \vdash \sigma \leq \sigma'}{\Delta \vdash \tau \Rightarrow \sigma \leq \tau' \Rightarrow \sigma'}$$

(Sub All)

$$\frac{\Delta, X \leq \delta \vdash \tau \leq \sigma}{\Delta \vdash \forall_{X \leq \delta}\, \tau \leq \forall_{X \leq \delta}\, \sigma}$$

(Sub RecL)

$$\frac{\Delta \vdash \tau[X \leftarrow \mu X.\tau] \leq \sigma}{\Delta \vdash \mu X.\tau \leq \sigma}$$

(Sub RecR)

$$\frac{\Delta \vdash \tau \leq \sigma[X \leftarrow \mu X.\sigma]}{\Delta \vdash \tau \leq \mu X.\sigma}$$

Figure 6.7: Hints about subtyping.

function types, the kernel-Fun discipline for polymorphic types, and the unfolding of recursive types, we also have a rule that covariantly relates interfaces, in both depth and width, Rule (Sub Interface).

Notice that the relation between component types, Rule (Sub Component), is contravariant in the required ports and covariant in the provided ones. This rule, in combination with Rule (Sub Interface) that relates the sets of required and provided services, defines the relation described before (Section 6.1) which allows for the use of components that require at most the same services than expected, possibly being more generic, and provide at least the same services than expected, possibly implementing more specific services. Technically, this results in a subtyping relation similar to the one found in arrow types, cf. Rule (Sub Fun). This is not at all surprising as components may be seen as service transformers: they implement a set of "resulting" services based on an instantiation of a set of "parameter" services. The difference here, besides the nature of the values, is that type τ and type σ are both object types declaring services in named ports which have a distinguished existence in the architecture of components.

Notice that configurator and record types are not mentioned in the rules of Figure 6.7.

In the case of record values, inclusion polymorphism is possible by means of their type coercion to interface types, by Rule (Val Interface), and then by subtyping. This allows for a record value, typed by a record type, to be recognised and used in contexts where only an interface type is specified. Record values, while allowing update operations, i.e. typed by a record type, can only be seen using a single type description.

Configurator types are related by equivalence only, which is defined in Rule (Sub Equal). Although not considered here, we describe, in section 6.4, a possible subtyping relation on configurator types which uniformly expresses the safe substitution of configurator values.

The result we extract from Chapter 5 is that an inductively defined relation, such as the one inductively defined by the Rules in Figure 6.7, does not include the relation on the infinite trees resulting from recursively defined types. A coinductive definition is therefore necessary to cover the relation between infinite unfoldings of recursive types.

In consequence of this, and as pointed out in [78], the general transitivity property, represented in Rule (Sub Trans), cannot be used in the definition of such a coinductive relation. The elimination of such a rule, obtaining nevertheless a relation closed under transitivity, is essential to design any subtyping algorithm.

We replace the explicit transitivity rule by a more specific form of transitivity, restricted to type variables, which we had already used in Chapter 5 for the case of kernel-Fun:

$$\text{(Sub TransVar)}\qquad \frac{\Delta \vdash \tau \leq \sigma \qquad X \leq \tau \in \Delta}{\Delta \vdash X \leq \sigma}$$

Based on the cases expressed by the rules in Figure 6.7 and Rule (Sub TransVar), we define the intended subtyping relation for $\lambda^{\leq}_{\mathcal{X}}$ as the greatest fixed point of a monotonic function on sets of judgements. Given the representation (Δ, τ, σ) for judgements of the form $\Delta \vdash \tau \leq \sigma$ and the set $\mathcal{J} \triangleq \mathcal{D}^{\leq} \times \mathcal{T}^{\leq}_{\mathcal{X}} \times \mathcal{T}^{\leq}_{\mathcal{X}}$ we define the following function:

Definition 6.12 (Generating function)**.** *The* generating function *is the map $S \in \mathcal{P}(\mathcal{J}) \to \mathcal{P}(\mathcal{J})$ defined by:*

$$\begin{array}{rcll}
S(\mathfrak{R}) & = & \{(\Delta;\tau;\tau) \mid \Delta \vdash \tau\ \mathsf{ok}\} & \textit{(Sub Equal)} \\
 & \cup & \{(\Delta;\tau;\top) \mid \Delta \vdash \tau\ \mathsf{ok}\} & \textit{(Sub Top)} \\
 & \cup & \{(\Delta;\tau \to \sigma;\tau' \to \sigma') \mid (\Delta;\tau';\tau),(\Delta;\sigma;\sigma') \in \mathfrak{R}\} & \textit{(Sub Fun)} \\
 & \cup & \{(\Delta;\{\ell_i : \tau_i{}^{i\in 1..m}\};\{\ell_i : \sigma_i{}^{i\in 1..n}\}) \mid n \leq m \textit{ and } (\Delta;\tau_i;\sigma_i)\ \ \forall i \in 1..n\} & \textit{(Sub Interface)} \\
 & \cup & \{(\Delta;\tau \Rightarrow \sigma;\tau' \Rightarrow \sigma') \mid (\Delta;\tau';\tau),(\Delta;\sigma;\sigma') \in \mathfrak{R}\} & \textit{(Sub Component)} \\
 & \cup & \{(\Delta;X;\sigma) \mid (\Delta;\tau;\sigma) \in \mathfrak{R} \textit{ and } X \leq \tau \in \Delta\} & \textit{(Sub TransVar)} \\
 & \cup & \{(\Delta;\forall_{X\leq\delta}\,\tau;\forall_{X\leq\delta}\,\sigma) \mid (\Delta, X \leq \delta;\tau;\sigma) \in \mathfrak{R}\} & \textit{(Sub All)} \\
 & \cup & \{(\Delta;\tau;\mu X.\sigma) \mid (\Delta;\tau;\sigma[X \leftarrow \mu X.\sigma]) \in \mathfrak{R}\ \} & \textit{(Sub RecR)} \\
 & \cup & \{(\Delta;\mu X.\tau;\sigma) \mid (\Delta;\tau[X \leftarrow \mu X.\tau];\sigma) \in \mathfrak{R} \textit{ and } \sigma \not\equiv \mu X.\sigma'\ \} & \textit{(Sub RecL)}
\end{array}$$

Notice that the cases of S defined above correspond to the rules in Figure 6.7 (with Rule (Sub TransVar) instead of Rule (Sub Trans)). Notice also that the least-fixed-point of this monotonic function is the inductive relation defined by the subtyping rules in Figure 5.1.

We define our subtyping relation as greatest-fixed-point of S, written νS. We the say that:

Definition 6.13 (Subtyping)**.** $\Delta \vdash \tau \leq \sigma \triangleq (\Delta, \tau, \sigma) \in \nu S.$

(Val Subsumption)

$$\frac{\Delta \vdash e : \sigma \quad \Delta \vdash \sigma \leq \tau}{\Delta \vdash e : \tau}$$

(Val Type Abstraction)

$$\frac{\Delta, X \leq \tau \vdash e : \sigma}{\Delta \vdash \Lambda X \leq \tau . e : \forall_{X \leq \tau}\, \sigma}$$

(Val Type Application)

$$\frac{\Delta \vdash e : \forall_{X \leq \tau}\, \sigma \quad \Delta \vdash \tau' \leq \tau}{\Delta \vdash e\langle \tau' \rangle : \sigma[X \leftarrow \tau']}$$

Figure 6.8: Typing rules for $\lambda_{\chi}^{\leq}$.

(Comp Requires)

$$\Delta \vdash (\mathsf{requires}\ \ell : \tau) : \emptyset \Longrightarrow \{\ell \bullet \tau, \ell \lhd \tau\}$$

(Comp Provides)

$$\Delta \vdash (\mathsf{provides}\ \ell : \tau) : \emptyset \Longrightarrow \{\ell \circ \tau, \ell \rhd \tau\}$$

(Comp Plug)$^{\leq}$

$$\frac{\Delta \vdash \tau_1 \leq \tau_2}{\Delta \vdash \mathsf{plug}\ (\pi_1 : \tau_1)\ \mathsf{into}\ (\pi_2 : \tau_2) : (\{\pi_2 \circ \tau_2, \pi_1 \bullet \tau_1\} \Longrightarrow \{\pi_1 \bullet \tau_1\})}$$

(Comp Sequence) $(K' \# K'', K' \# K''')$

$$\frac{\Delta \vdash e_1 : K \Longrightarrow K', K_c \quad \Delta \vdash e_2 : K_c, K'' \Longrightarrow K'''}{\Delta \vdash (e_1 ; e_2) : K, K'' \Longrightarrow K', K'''}$$

(Comp Uses) $(\tau = \{\ell_i^r : \tau_i\ ^{i \in 1..n}\},\ \sigma = \{\ell_j^p : \sigma_j\ ^{j \in 1..m}\})$

$$\frac{\Delta \vdash e : \tau \Rightarrow \sigma}{\Delta \vdash x[e : \tau \Rightarrow \sigma] : \emptyset \Longrightarrow \left\{x \bullet \sigma, x.\ell_i^r \circ \tau_i\ ^{i \in 1..n}, x.\ell_j^p \bullet \sigma_j\ ^{j \in 1..m}\right\}}$$

(Comp Method Block) $(I = \{\ell_i' : \tau_i'\ ^{i \in 1..m}\},\ K = \{\ell_i' \bullet \tau_i'\ ^{i \in 1..n}\})$

$$\frac{|\Delta|, \ell_i' : \tau_i'\ ^{i \in 1..m}, x : \{\!|\ell_i : \tau_i\ ^{i \in 1..n}|\!\} \vdash e_i : \tau_i \quad \forall_{i \in 1..n}}{\Delta \vdash x_I[\ell_i : \tau_i = e_i\ ^{i \in 1..n}] : K \Longrightarrow K, \{x \bullet \{\ell_i : \tau_i\ ^{i \in 1..n}\}\}}$$

Figure 6.9: Typing rules for $\lambda_{\chi}^{\leq}$ (part 2).

This is a declarative definition of the subtyping relation, later in this chapter we define an algorithm to determine whether a given judgement is in the subtyping relation or not, thus giving it an operational definition.

Typing

We next define the typing relation of $\lambda_{\chi}^{\leq}$ by extending the typing relation of $\lambda_{\chi}^{\leq}$ and using the subtyping relation in a subsumption rule which uniformly applies to all expressions.

Definition 6.14 (Typing relation). *The judgement $\Delta \vdash e : \tau$ is valid if it is derivable by the rules in Figures 2.11, 3.12, 6.9, 4.8, and 6.8.*

We follow by explaining the rules in Figures 6.8, and 6.9. Rule (Val Subsumption), as expected, allows the usage of a value in any context where a supertype is expected.

$$\text{(Wrong Type App)}\quad \frac{e;S \downarrow v;S' \quad v \neq \Lambda X \leq \tau'.e'}{e\langle\tau\rangle;S \downarrow \mathsf{wrong};S'}$$

Figure 6.10: Error trapping rule for $\lambda_\chi^\leq$.

The typing of type abstractions, defined in Rule (Val Type Abstraction), and the typing of type application expressions, defined in Rule (Val Type Application), follow along standard lines. In a type abstraction $\Lambda X \leq \tau.e$, the body e is typed in a context where the declared variable X has its corresponding bound τ, and the application rule is only applicable if the type argument τ' in expression $e\langle\tau'\rangle$ is a subtype of the declared bound. The resulting type is the type of the expression where the type variable (the type argument) is replaced by the actual type argument.

With regard to composition operations we change the typing of plug expressions in order to consider subtyping between the port types. In this type system, the source and target of a plug expression must be compatible by subtyping instead of type equivalence, Rule (Comp Plug)$^\leq$.

This completes the definition of the type system for $\lambda_\chi^\leq$ which consists of adding standard constructs, that deal with type abstraction and recursive types, to the type system of λ_ρ. We now show that the resulting language is sound and has a decidable type checking problem.

6.2.1 Type safety

The type safety result we prove next extends the previous type safety results in the sense that it allows the subsumption in the values resulting from evaluating well-typed expressions, i.e. the type of the value can be a subtype of the statically assigned type. We use the same technique as before. We enunciate and prove correct a subject reduction theorem that captures this property on the results of an operational semantics extended with a distinguished value wrong and an evaluation rule that captures errors in type applications, Figure 6.10.

Definition 6.15 (Extended Evaluation). *Let $e \in \lambda_\chi^\leq$ and a heap S such that $(e;S)$ is a valid configuration. The evaluation relation of an expression e to a value v, written $e;S \downarrow v;S'$ is defined inductively by the rules in Figures 2.8, 2.12, 4.4, 4.5, 4.6, 4.7, 3.14, 4.9, 6.4, and 6.10.*

The intermediate result which introduces the computational and structural invariants on well-typed expressions, as Lemma 3.29 for λ_χ and 4.12 for λ_ρ, must now incorporate subsumption of resulting values. We must first rewrite the definition of the typing of heaps to allow locations to map to values yielding a subtype of what is expected.

Definition 6.16 (Typing of Heaps). *For any typing environment Γ and heap S, we say that Γ* types *S if* $\mathrm{Dom}(\Gamma) \subseteq \mathrm{Dom}(S)$ *and* $\forall\, l \in \mathrm{Dom}(S)$ *we have that*

1. *l is `undefined`,*

2. *l `refers-to` nil, or*

3. *l `refers-to` v and $\Gamma \vdash v : \tau$ with $\Gamma \vdash \tau' \leq \tau$.*

Furthermore, we need to introduce new lemmas for weakening, substitution of values and types in expressions and types; we also must state the transitivity property of the subtyping relation. Remember that transitivity is not explicit in the relation, and that the closure of the subtyping relation under transitivity is a side effect of its definition.

Lemma 6.17 (Transitivity). *For all typing environments Δ, Δ', and types $\tau, \sigma, \delta \in \mathcal{T}_{\mathcal{X}}^{\leq}$:*
If $\Delta \vdash \tau \leq \delta$ and $\Delta \vdash \delta \leq \sigma$ then $\Delta \vdash \tau \leq \sigma$.

Proof. This property is expressed in Lemma 5.17 for kernel-Fun. Remember that the two type languages differ only by two type constructs, component and interface types, and that these constructs are uniform under transitivity. One has the exact same structure as function types and the other is a labelled product type.

So, transitivity on these type constructs depends directly on the transitivity on their subexpressions. The proof of this lemma is therefore similar to that of Lemma 5.17. It is based on an extended relation that includes the closure of the subtyping relation under transitivity and narrowing. The proof is done by coinduction on the definition of the generating function. □

The preservation of types under the substitution of types is defined for both the substitution of type variables in type expressions and in terms.

Lemma 6.18 (Substitution of type variables in types). *For all typing environments Δ, Δ', and types $\tau, \tau', \sigma, \sigma' \in \mathcal{T}_{\mathcal{X}}^{\leq}$:*
If $\Delta, X \leq \tau, \Delta' \vdash \sigma \leq \sigma'$ and $\Delta \vdash \tau' \leq \tau$ then $\Delta, \Delta' \vdash \sigma[X \leftarrow \tau'] \leq \sigma'[X \leftarrow \tau']$.

Proof. By proving coinductively that the subtyping relation is closed under substitution of type variables. The proof for this property is similar to the proof of Lemma 5.18 for kernel-Fun. Remember that substitution propagates uniformly through the new type constructs. □

Lemma 6.19 (Substitution of type variables in terms). *For all typing environments Δ, Δ', all expressions $e \in \lambda_{\mathcal{X}}^{\leq}$, and types $\tau, \tau', \sigma, \sigma' \in \mathcal{T}_{\mathcal{X}}^{\leq}$:*
If $\Delta, X \leq \tau, \Delta' \vdash e : \sigma$ and $\Delta \vdash \tau' \leq \tau$ then $\Delta, \Delta' \vdash e[X \leftarrow \tau'] : \sigma'$ with $\Delta \vdash \sigma' \leq \sigma$.

Proof. By induction on the height of the derivation and by case analysis of the last rule used. □

Lemma 6.20 (Weakening)**.** *For all typing environments Δ, Δ', all expressions $x, e \in \lambda_{\chi}^{\leq}$, and types $\tau \in \mathcal{T}_{\chi}^{\leq}$:*
If $\Delta, \Delta' \vdash e : \tau$ and $x \notin Dom(\Delta')$ then $\Delta, x : \tau', \Delta' \vdash e : \tau$.

Proof. By induction on the height of the derivation and by case analysis of the last rule used. □

Lemma 6.21 (Substitution)**.** *For all typing environments Δ, Δ', all expressions $e \in \lambda_{\chi}^{\leq}$, and types $\tau, \sigma, \sigma' \in \mathcal{T}_{\chi}^{\leq}$:*
If $\Delta, x : \tau, \Delta' \vdash e : \sigma$ and $\Delta \vdash v : \tau'$ with $\Delta \vdash \tau' \leq \tau$ then $\Delta, \Delta' \vdash e[x \leftarrow v] : \sigma'$ with $\Delta \vdash \sigma' \leq \sigma$. Moreover, if e and v are record-based then $e[x \leftarrow v]$ is also record-based.

Proof. By induction on the height of the derivation and by case analysis of the last rule used. □

We now enunciate the following general lemma that proves subject reduction and captures the subsumption of the result with relation to the expected type.

Lemma 6.22 (Subject Reduction)**.**

1. *Let $(e; S)$ be a valid configuration in $\lambda_{\chi}^{\leq} \backslash \{\mathsf{nil}\}$ such that $\mathsf{nil}(S) = \emptyset$ and let Γ be a typing environment typing S such that e and S are record-based with relation to Γ.*

   If $\Gamma \vdash e : \tau$ and $(e; S \downarrow v; S')$ then

   a) there is a Γ' that extends Γ and types S',

   b) $\Gamma' \vdash v : \tau$ such that $\Delta \vdash \tau' \leq \tau$,

   c) v is either an abstraction, a component, a configurator, or a location that is either `undefined` or `refers-to` a record,

   d) v and S' are record-based with relation to Γ, and

   e) $\mathsf{nil}(S') = \emptyset$.

2. *Let c be an expression such that $\Gamma \vdash c : K \Longrightarrow K'$, let S be a heap such that, for some set $X \in Dom(S)$, $\mathsf{nil}(S) \subseteq \{\mathrm{select}_S(s, \pi) \mid (\pi : \tau) \in K_{\triangleleft} \cup K_{\circ}\} \uplus X$ and Γ types S.*

   Let s be a partially linked object s such that it complies with K and its partially linked object type is $[\![R \oplus K_{\triangleleft} \Rightarrow P \oplus K_{\triangleright}]\!]$ and s and S are record-based with relation to Γ:

   If $s; c; S \Downarrow s'; S'$ then

   a) there is Γ' typing S' and extending Γ,

   b) s' is a partially linked object that extends s and complies with K'. Its partially linked object type is $[\![R \oplus K'_{\triangleleft} \Rightarrow P \oplus K'_{\triangleright}]\!]$, and

   c) s' and S' are record-based with relation to Γ', and

   d) $\mathsf{nil}(S') \subseteq \{\mathrm{select}_{S'}(s, \pi) \mid (\pi : \tau) \in K'_{\triangleleft} \cup K'_{\circ}\} \uplus X$.

Proof Sketch. The proof is carried out by induction on the two cases of the lemma. On the first case, we prove it by induction on the size of the evaluation derivations and by analysis of the last rule used. We use the second case when needed and its conditions are met. We show, in the cases that evaluate to wrong, that these rules are never applicable. We here present the cases for the new constructions and those where the effect of subtyping is most relevant. (for the complete proof see appendix A on page 206).

***Case:* (Eval Value)**

The conclusions of the theorem hold with $\Gamma' = \Gamma$, $S' = S$, and $\Gamma \vdash v : \tau$ with $\Gamma \vdash \tau \leq \tau$ by definition of subtyping, Definition 6.13. We also have that $\mathsf{nil}(S') = \emptyset$ and that both v and S are record-based.

***Case:* (Eval Application)$^{\tau}$**

In this case, with $e_1(e_2); S \downarrow v; S'$, we know by Rule (Eval Application), that: a) $e_1; S \downarrow \lambda x : \tau.e; S'$, b) $e_2; S' \downarrow v_2; S''$, and c) $e[x \leftarrow v_2]; S'' \downarrow v; S'''$. On the typing derivation, the only plausible last rule for $\Gamma \vdash e_1(e_2) : \sigma$ is Rule (Val Application), with the premises: d) $\Gamma \vdash e_1 : \tau \rightarrow \sigma$ and e) $\Gamma \vdash e_2 : \tau$.

Taking d) and a), by induction hypothesis we conclude that there is a Γ' extending Γ and typing S', such that f) $\Gamma' \vdash \lambda x : \tau.e : \tau' \rightarrow \sigma'$ with $\Gamma' \vdash \tau' \rightarrow \sigma' \leq \tau \rightarrow \sigma$. By Lemma 3.24 (weakening) on e) we conclude that $\Gamma' \vdash e_2 : \tau$, which together with b), by induction hypothesis, there is a Γ'' extending both Γ' and Γ and typing S'' such that g) $\Gamma'' \vdash v_2 : \tau''$ with $\Gamma'' \vdash \tau'' \leq \tau$ and $\mathsf{nil}(S'') = \emptyset$. From f), by Lemma 3.24 (weakening) and Rule (Val Abstraction) we have that $\Gamma'', x : \tau \vdash e : \sigma$ and by Lemma 3.28 (substitution) with g) we have that h) $\Gamma'' \vdash e[x \leftarrow v_2] : \sigma'$ with $\Gamma'' \vdash \sigma' \leq \sigma$. Taking h) and c), by induction hypothesis we have that there is Γ''' extending Γ'' and typing S''', such that v is typed $\Gamma''' \vdash v : \sigma$ and v is either an abstraction or a location that is either ***undefined*** or ***refers-to*** a record, and $\mathsf{nil}(S''') = \emptyset$. Notice that v and S''' are record-based.

The application of induction hypothesis to the different premises and the application of Lemma 3.28 ensures that v and S''' are record-based.

***Case:* (Eval Type Application)**

If the last rule is (Eval Type Application) the for the expression $e\langle\tau\rangle$ we have the typing judgement, whose premise must be a) $\Delta \vdash e\langle\tau\rangle : \sigma[X \leftarrow \tau]$ and the evaluation judgement b) $e\langle\tau\rangle; S \downarrow v; S''$. The premises for b) are, c) $e; S \downarrow \Lambda X \leq \tau.e'; S'$ and d) $e'[X \leftarrow \tau']; S' \downarrow v; S''$. By induction hypothesis we know that there is a typing environment Γ' which extends Γ and types S' and $\Gamma' \vdash \Lambda X \leq \tau.e' : \forall_{X \leq \tau}\, \sigma'$ such that $\Gamma' \vdash \forall_{X \leq \tau}\, \sigma' \leq \forall_{X \leq \tau}\, \sigma$. From the polymorphic case of S (Definition 6.12) we know that $\Gamma', X \leq \tau \vdash \sigma' \leq \sigma$. By Rule (Val Type Abstraction) on a) we know that $\Gamma', X \leq \tau \vdash e' : \sigma$. Since $\Gamma \vdash \tau' \leq \tau$, by Lemma 6.20 (weakening) we have

$\Gamma' \vdash \tau' \leq \tau$, and by Lemma 6.19 (substitution) we have that e) $\Gamma' \vdash e'[X \leftarrow \tau'] : \sigma'[X \leftarrow \tau']$ and by Lemma 6.18 (substitution) we have that $\Gamma' \vdash \sigma'[X \leftarrow \tau'] \leq \sigma[X \leftarrow \tau']$.

From e) and d), by induction hypothesis, we have that there is a Γ'' that extends Γ and types S'', such that $\Gamma'' \vdash v : \sigma''$ with $\Gamma'' \vdash \sigma'' \leq \sigma'[X \leftarrow \tau']$ and v is one of the possible values admitted in the lemma. The induction hypothesis on both cases indicates that $\mathsf{nil}(S'') = \emptyset$. Notice that v and S''' are record-based. □

We now enunciate the main type safety result as corollary of Lemma 6.22. As in previous type safety results, the final subject reduction theorem isolates the fundamental type preservation property of $\lambda_\chi^{\leq}$. Lemma 6.22 is needed to ensure that types are preserved in compositions as well.

Theorem 6.23 (Subject Reduction). *Let $(e;S)$ be a valid configuration of $\lambda_\chi^{\leq}\backslash\{\mathsf{nil}\}$ and S a heap not containing any* nil *value, let Γ be a typing environment typing the heap S such that e and S are record-based with relation to Γ.*
If $\Gamma \vdash e : \tau$ and $e; S; \Gamma \downarrow v; S'; \Gamma'$ then

a) Γ' extends Γ and types S',

b) $\Gamma' \vdash v : \tau'$ with $\Gamma' \vdash \tau' \leq \tau$, and

c) v is either an abstraction, a component, a configurator, or a location that is either `undefined` or `refers-to` a record.

d) If v is a component or configurator value then $FL(v) = \emptyset$, and

e) $\mathsf{nil}(S') = \emptyset$.

Proof. This theorem results directly from the first case of Lemma 6.22 and Lemma 3.25. □

We conclude that $\lambda_\chi^{\leq}$ conservatively extends λ_ρ, i.e. there are more λ_ρ programs accepted by the type system presented here due to the introduction of subtyping. Theorem 6.23 implies that, besides uniformly extending the typing of computational expressions, $\lambda_\chi^{\leq}$ also extends the notion of architecture soundness by means of subsumption in the introduction of internal components and of subtyping between port types in **plug** operations. Notice that when a component is introduced in a composition it may require less ports than expected (by subsumption). In this case, we use the type expression provided in the expression $x[e:\tau]$ to correctly type the connections and rely on the operational semantics to ignore the lack of a target port (already predicted in the semantics of λ_χ).

$$\text{(X-Var)}\quad \frac{X \leq \sigma \in \Delta \quad \Delta \vdash \sigma \Uparrow \tau}{\Delta \vdash X \Uparrow \tau} \qquad \text{(X-Default)}\quad \frac{\tau \notin Dom(\Delta)}{\Delta \vdash \tau \Uparrow \tau}$$

Figure 6.11: Exposure rules for type variables.

6.3 Typing Algorithm

It is our claim that $\lambda^{\leq}_{\chi}$ has all the essential elements to be a quite general component-based object-oriented language. We have, so far, defined a precise semantics and a typing relation. In this section, we define a typing algorithm which proves the decidability of the typing relation.

Typing algorithms designed out of inductive systems are generally defined by backward interpretation of typing rules. However, there are some rules in Definition 6.14 for which a backward interpretation is not feasible. Notably, subsumption, explicit in the rule

$$\frac{\Delta \vdash e : \sigma \quad \Delta \vdash \sigma \leq \tau}{\Delta \vdash e : \tau}$$

cannot be included in the algorithm. The result of this rule (the judgement $\Delta \vdash e : \tau$) does not have, in general, a single instantiation for τ. The same happens with the explicit transitivity rule, Rule (Sub Trans) in Figure 6.7. The elimination of transitivity rule on subtyping was already justified earlier in the context of the subtyping coinductive definition. To achieve the same typing relation without using the indeterministic subsumption rule, we replace it by carefully placing subtype checks in the premises of other rules. This transformation allows for the deterministic interpretation of all other rules and therefore permits the design of an algorithm to compute the type of an expression with relation to a typing environment. In the case of the rules that need to inspect the internal structure of a type (the so-called destructive rules), as it is the case of the rule for record selections, Rule (Val Select), we use a special notion of variable exposure, which is an explicit kind of deterministic subsumption to the lowest, concrete supertype.

Definition 6.24. *Type σ is the exposure of type τ if the judgement $\Delta \vdash \tau \Uparrow \sigma$ is derivable by the rules in Figure 6.11.*

Intuitively, the exposure of a type is, in the case of type variables, its nearest non-variable bound, and it is the type itself otherwise. When used in type variables, it exposes the common structure of all the types that may instantiate them. It replaces the subsumption relation for type variables.

Another important part of the typing algorithm is the verification of subtyping judgements. This is obtained by direct application of the results of Chapter 5 to the generating function in

```
$Subtyping(A,(\Delta,\tau,\sigma)) =$
    if $(\Delta,\tau,\sigma) \in^{\simeq} A$ then $A$
    else let $A_0 = A \cup \{(\Delta,\tau,\sigma)\}$ in
    if $\tau \equiv \sigma$ then $A$
    else if $\sigma \equiv \top$ then $A$
    else if $\tau \equiv \tau' \rightarrow \tau''$ and $\sigma \equiv \sigma' \rightarrow \sigma''$ then
      let $A_1 = Subtyping(A_0,(\Delta,\tau'',\sigma''))$ in $Subtyping(A_1,(\Delta,\sigma',\tau'))$
    else if $\tau \equiv \tau' \Rightarrow \tau''$ and $\sigma \equiv \sigma' \Rightarrow \sigma''$ then
      let $A_1 = Subtyping(A_0,(\Delta,\tau'',\sigma''))$ in $Subtyping(A_1,(\Delta,\sigma',\tau'))$
    else if $\tau \equiv \{\ell_i : \tau_i{}^{i\in 1..n}\}$ and $\sigma \equiv \{\ell_j : \sigma_j{}^{j\in 1..m}\}$ then
      if $n \geq m$ then $(\forall_{i\in 1..m}$ let $A_i = Subtyping(A_{i-1},(\Delta,\tau_i,\sigma_i)))$ in $A_m$
    else if $\tau \equiv X$ then $Subtyping(A_0,(\Delta,\Delta(X),\sigma))$
    else if $\tau \equiv \forall_{X\leq\delta}\,\tau'$ and $\sigma \equiv \forall_{X\leq\delta}\,\sigma'$ then $Subtyping(A_0,(\Delta, X \leq \delta;\tau';\sigma'))$
    else if $\tau \equiv \mu X.\tau'$ then $Subtyping(A_0,(\Delta,\tau'[X\leftarrow\tau],\sigma))$
    else if $\sigma \equiv \mu X.\sigma'$ then $Subtyping(A_0,(\Delta,\tau,\sigma'[X\leftarrow\sigma]))$
    else $fail$
```

Figure 6.12: Subtyping algorithm.

Definition 6.12. The result is basically an extension of the algorithm of Definition 5.24 with the new type constructors for component and interface types.

Definition 6.25 (Subtyping algorithm)**.** *$Subtyping(A,(\Delta,\tau,\sigma))$ is defined by the procedure described in Figure 6.12.*

Besides the subtyping algorithm and the exposure of type variables we need other deterministic operations to access the types of fields in record and interface types, and of variables in typing environments.

Definition 6.26 (Field lookup)**.**

$$\begin{aligned}
\text{lookup}(\ell,\{\ldots,\ell:\tau,\ldots\}) &\triangleq \tau\\
\text{lookup}(\ell,\{\!|\ldots,\ell:\tau,\ldots|\!\}) &\triangleq \tau\\
\text{lookup}(\ell,\mu X.\tau) &\triangleq \text{lookup}(\ell,\tau[X\leftarrow\mu X.\tau])
\end{aligned}$$

We indistinguishably treat records and interfaces and look into the unfolding of recursive types in search for field labels. Note that we assume that recursive types are contractive and therefore there is at most one unfolding operation involved in this operation. Notice that the similarity relation on tuples ($\simeq$) used in the algorithm above is defined in Chapter 5, Definition 5.20, in such a way that it can be reused here. Remember that $t \in^{\simeq} A \triangleq \exists u \in A.t \simeq u$.

Hence, combining all these elements together we define the typing algorithm based on the backward interpretation of the rules in Figures 6.13 and 6.14 to obtain an instantiation of τ from an input Δ and e, in the judgement $\Delta \vdash_a e : \tau$.

(Val Var)^a

$$\frac{x:\tau \in \Delta}{\Delta \vdash_a x:\tau}$$

(Val Abstraction)^a

$$\frac{\Delta, x:\tau \vdash_a e:\sigma}{\Delta \vdash_a \lambda x:\tau.e:\tau \to \sigma}$$

(Val Application)^a

$$\frac{\Delta \vdash_a e_1:\tau \to \sigma \quad \Delta \vdash_a e_2:\tau' \quad \Delta \vdash \tau' \leq \tau}{\Delta \vdash_a e_1(e_2):\sigma}$$

(Val Record)^a

$$\frac{\Delta \vdash_a e_i:\tau_i}{\Delta \vdash_a \{\ell_i = e_i{}^{i\in 1..n}\} : \{\!|\ell_i:\tau_i{}^{i\in 1..n}|\!\}}$$

(Val Select)^a

$$\frac{\Delta \vdash_a e:\sigma \quad \Delta \vdash \sigma \Uparrow \sigma' \quad \mathrm{lookup}(\ell,\sigma') = \tau}{\Delta \vdash_a e.\ell:\tau}$$

(Val Assign)^a

$$\frac{\Delta \vdash_a e_1:\sigma \quad \Delta \vdash \sigma \Uparrow \{\!|\ell_i:\tau_i{}^{i\in 1..n}|\!\} \quad \mathrm{lookup}(\ell, \{\!|\ell_i:\tau_i{}^{i\in 1..n}|\!\}) = \tau \quad \Delta \vdash_a e_2:\tau' \quad \Delta \vdash \tau' \leq \tau}{\Delta \vdash_a e_1.\ell := e_2:\tau}$$

(Val Compose)^a $(K_\circ = \emptyset)$

$$\frac{\Delta \vdash_a e:\emptyset \Longrightarrow K}{\Delta \vdash_a \mathsf{compose}\ e:K_\triangleleft \Rightarrow K_\triangleright}$$

(Val New)^a

$$\frac{\Delta \vdash_a e:\{\ell_i:\tau_i{}^{i\in 1..n}\} \Rightarrow \sigma \quad \Delta \vdash_a e_i:\tau'_i \quad \Delta \vdash \tau'_i \leq \tau_i \quad \forall_{i\in 1..n}}{\Delta \vdash_a \mathsf{new}\ e\ \mathsf{with}\ \ell_i := e_i{}^{i\in 1..n}:\sigma}$$

(Val Reconfig) $\big(K'_\circ = \emptyset,\ K'_\triangleright \# I,\ K'_\triangleleft = \{\ell_i:\sigma_i{}^{i\in 1..n}\}\big)$

$$\frac{\begin{array}{c}\Delta \vdash_a e_1:K \Longrightarrow K' \quad \Delta \vdash_a e_2:I \quad \Delta \vdash_a e'_i:\sigma'_i \quad \Delta \vdash \sigma'_i \leq \sigma_i \ \forall_{i\in 1..n} \\ \Delta, x:I \oplus K'_\triangleright \vdash_a e_3:\delta \quad \Delta, x:I \vdash_a e_4:\delta\end{array}}{\Delta \vdash_a \mathsf{reconfig}\ x = e_1[e_2]\ \mathsf{with}\ \ell_i := e'_i{}^{i\in 1..n}\ \mathsf{in}\ e_3\ \mathsf{else}\ e_4:\delta}$$

(Val Type Abstraction)

$$\frac{\Delta, X \leq \tau \vdash_a e:\sigma}{\Delta \vdash_a \Lambda X \leq \tau.e:\forall_{X\leq\tau}\,\sigma}$$

(Val Type Application)

$$\frac{\Delta \vdash_a e:\forall_{X\leq\tau}\,\sigma \quad \Delta \vdash \tau' \leq \tau}{\Delta \vdash_a e\langle\tau'\rangle:\sigma[X \leftarrow \tau']}$$

Figure 6.13: Algorithmic typing rules for $\lambda^{\leq}_{X}$.

Definition 6.27 (Typing algorithm)**.** *The type of an expression with relation to a typing environment is given by* typing(δ, e)*, defined by the backward interpretation of the rules in Figures 6.13 and 6.14.*

The type of a closed expression e is then given by a procedure typing$(\emptyset, e)$ defined by backward interpretation of the typing rules as illustrated in Figure 6.15. The procedure is not shown in full but can be easily completed by transcribing the rules in Figures 6.13 and 6.14 in a goal directed way. Notice that it builds a type for an expression based on the results of recursive calls on its subexpressions. Notice also that we have omitted Rule (Val Interface), which makes record types acceptable as the subject of a selection and incorporated this subsumption in the lookup operation which indistinguishably operates on records and interfaces.

(Comp Requires)^a $$\Delta \vdash_a (\textsf{requires}\ \ell : \tau) : \emptyset \Longrightarrow \{\ell \bullet I, \ell \lhd \tau\}$$

(Comp Provides)^a $$\Delta \vdash_a (\textsf{provides}\ \ell : \tau) : \emptyset \Longrightarrow \{\ell \circ \tau, \ell \rhd \tau\}$$

(Comp Plug)^a
$$\frac{\Delta \vdash \tau_1 \leq \tau_2}{\Delta \vdash_a \textsf{plug}\ (\pi_1 : \tau_1)\ \textsf{into}\ (\pi_2 : \tau_2) : (\{\pi_2 \circ \tau_2, \pi_1 \bullet \tau_1\} \Longrightarrow \{\pi_1 \bullet \tau_1\})}$$

(Comp Sequence)^a $(K' \# K'', K' \# K''')$
$$\frac{\Delta \vdash_a e_1 : K \Longrightarrow K', K_c \qquad \Delta \vdash_a e_2 : K_c, K'' \Longrightarrow K'''}{\Delta \vdash_a (e_1; e_2) : K, K'' \Longrightarrow K', K'''}$$

(Comp Uses)^a $(\tau = \{\ell_i : \tau_i\ ^{i \in 1..k}\},\ \sigma = \{\ell'_j : \sigma_j\ ^{j \in 1..m}\})$
$$\frac{\Delta \vdash_a e : \tau' \Rightarrow \sigma' \quad \Delta \vdash \tau' \Rightarrow \sigma' \leq \tau \Rightarrow \sigma}{\Delta \vdash_a x[e : \tau \Rightarrow \sigma] : \emptyset \Longrightarrow \{x \bullet \sigma, x.\ell_i \circ \tau_i\ ^{i \in 1..n}, x.\ell'_j \bullet \sigma_j\ ^{j \in 1..m}\}}$$

(Comp Method Block)^a $(I = \{\ell'_i : \tau'_i\ ^{i \in 1..m}\},\ K = \{\ell'_i \bullet \tau'_i\ ^{i \in 1..n}\})$
$$\frac{|\Delta|, \ell'_i : \tau'_i\ ^{i \in 1..m}, x : \{\!|\ell_i : \tau_i\ ^{i \in 1..n}|\!\} \vdash_a e_i : \tau'_i \quad \Delta \vdash \tau'_i \leq \tau_i \quad \forall_{i \in 1..n}}{\Delta \vdash_a x_K[\ell_i : \tau_i = e_i\ ^{i \in 1..n}] : K \Longrightarrow K, \{x \bullet \{\ell_i : \tau_i\ ^{i \in 1..n}\}\}}$$

Figure 6.14: Algorithmic typing rules for $\lambda_\chi^\leq$ (part 2).

Correctness

To prove that our typing algorithm, is correct with relation to the typing relation presented in section 6.2, we first prove that the subtyping algorithm is also correct with relation to the defined subtyping relation.

Lemma 6.28 (Correctness of *Subtyping*)**.** *For all $\Delta \in \mathcal{D}$, and types $\tau, \sigma \in \mathcal{T}_\leq$,*

1. $Subtyping(\emptyset, (\Delta, \tau, \sigma)) = A$ if and only if $\Delta \vdash \tau \leq \sigma$

2. $Subtyping(\emptyset, (\Delta, \tau, \sigma)) = fail$ if and only if $\Delta \nvdash \tau \leq \sigma$

Proof. This property is based on Lemma 5.39 where the definition of the relation is extended with the regular type constructors. The result of the algorithm in these cases is directly supported by the results of recursive calls and therefore the proof is identical to that of Lemma 5.39. □

We also need an auxiliary result that relates the exposure of a type variable and subtyping as follows.

Lemma 6.29 (Exposure and subtyping)**.** *For all $\Delta \in \mathcal{D}$, and types $\tau \in \mathcal{T}_\leq$,*
If $\Delta \vdash \tau \uparrow \tau'$ then $\Delta \vdash \tau \leq \tau'$.

```
                          typing$(\Delta, e) \triangleq$
(Val Var)$^a$               if $e = x$ then $\Delta(x)$
(Val Abstraction)$^a$       else if $e = \lambda x : \tau . e$ then let $\sigma = \text{typing}((\Delta, x : \tau), e)$ in $\tau \rightarrow \sigma$
(Val Application)$^a$       else if $e = e_1(e_2)$ then let $\delta = \text{typing}(\Delta, e_1)$ in
                              if $\delta = \tau \rightarrow \sigma$ then
                                let $\tau' = \text{typing}(\Delta, e_2)$ in $\text{Subtyping}(\emptyset, (\Delta, \tau', \tau)); \sigma$
                              else $fail$
(Val Record)$^a$            else if $e = \{\ell_i = e_i\ ^{i \in 1..n}\}$ then
                              $(\forall_{i \in 1..n}$ let $\tau_i' = \text{typing}(\Delta, e_i)$ in $\text{Subtyping}(\emptyset, (\Delta, \tau_i', \tau_i)))$; $\{\!|\ell_i : \tau_i\ ^{i \in 1..n}|\!\}$
(Val Select)$^a$            else if $e = e.\ell$ then let $\sigma = \text{typing}(\Delta, e)$ in
                              let $\sigma' = \text{expose}(\Delta, \sigma)$ in
                              let $\tau = \text{lookup}(\ell, \sigma')$in $\tau$
(Val Assign)$^a$            else if $e = e_1 := e_2$ then let $\sigma = \text{typing}(\Delta, e_1)$ in
                              let $\sigma' = \text{expose}(\Delta, \sigma)$ in
                              if $\sigma' = \{\!|\ell_i : \tau_i\ ^{i \in 1..n}|\!\}$ then
                                let $\tau = \text{lookup}(\ell, \sigma')$in
                                let $\tau' = \text{typing}(\Delta, e_2)$in $\text{Subtyping}(\emptyset, (\Delta, \tau', \tau)); \tau$
                              else $fail$
(Val Compose)$^a$           else if $e =$ compose $e$ then let $\tau = \text{typing}(\Delta, e)$ in
                              if $\tau = K \Longrightarrow K'$ $and$ $K = \emptyset$ $and$ $K'_\circ = \emptyset$ then $K'_\triangleleft \Rightarrow K'_\triangleright$
                              else $fail$
(Val New)$^a$               else if $e =$ new $e$ with $\ell_i^r := e_i\ ^{i \in 1..n}$ then
                              let $\tau = \text{typing}(\Delta, e)$ in
                              if $\tau = \{\ell_i : \tau_i\ ^{i \in 1..n}\} \Rightarrow \sigma$ then ...
                              else $fail$
                            ...
                          $\text{expose}(\Delta, \tau) \triangleq$ if $\tau = X$ then $\text{expose}(\Delta, \Delta(X))$ else $\tau$
```

Figure 6.15: The typing algorithm.

Proof. By induction on the height of the derivation. □

We finally enunciate and prove the correctness of the typing algorithm in the following theorem:

Theorem 6.30 (Soundness). *For all $\Delta \in \mathcal{D}$, expressions e and types $\tau \in \mathcal{T}_\leq$,*
If $\Delta \vdash_a e : \tau$ then $\Delta \vdash e : \tau$

Proof. By induction on the height of the derivation and by case analysis on the last rule used. Lemma 6.29 is applied when necessary. □

Theorem 6.31 (Completness). *For all $\Delta \in \mathcal{D}$, expressions e and types $\tau \in \mathcal{T}_\leq$,*
If $\Delta \vdash e : \sigma$ then $\Delta \vdash_a e : \tau$ with $\Delta \vdash \tau \leq \sigma$.

Proof. By induction on the height of the derivation and by case analysis on the last rule used. Lemma 5.17 is applied in the case of Rule (Val Subsumption). □

The property enunciated by this theorem is also known as the minimal typing property which ensures that the typing algorithm, which has at most one possible result for a given input, gives the more specific type for an expression with relation to a typing environment. By the subsumption property, many types are admissible for an expression, it is therefore important, as a completeness result, that the algorithm gives the least of them. In this way it allows, by subsumption, that any supertype of the result can be assigned to the expression.

This concludes the presentation of the type system and typing algorithm for $\lambda_{\chi}^{\leq}$. We now close the chapter with a proposal for a structural subtyping relation on configurator types.

6.4 Subtyping on Configurator Types

Structural subtyping relations support the safe substitution of values in a uniform way. Their definition is usually fairly intuitive and based on extensional type information (that describes values regardless of their internal structure). But, when looking to the intensional type information used in our model to type configurator values those intuitions get blurred by a multitude of degrees of freedom.

For configurator types, subtyping means that configurator values must introduce and connect architectural elements in such a way that it does not break the expected effect in a given well-typed context, i.e. if a configurator value yield a subtype of the expected type, the compositions where they are used, in that context, are still well-formed.

We analyse here some situations when a configurator type can be made more specific (in the subtyping sense) without disrupting the described effect. A subtyping relation including all these situation can the be defined using the rules depicted in Figure 6.16.

For instance, a subtype may indicate the addition of elements that are not going to be used in the context where a value of a supertype is expected, as in Rule (Sub • more provided). The demand for less available elements, described in Rule (Sub • less required) is another possibility. Symmetric relations can be found when analysing the unsatisfied resources, see Rules (Sub ∘ more required) and (Sub ∘ less provided). A configurator that requires some unsatisfied resource, and presumably satisfies it, can be used in a context where the need for that connection does not exist. Also, a configurator that introduces an unsatisfied resource can be replaced by one that does not. Notice that these substitutions may result in the making of void connections, which is already admissible in the original semantics (see discussion around Definition 3.7).

Additionally, the types associated to resources may also vary in the subtyping relation, either covariantly or contravariantly. The variance in the subtyping relation is influenced by

(Sub $\bullet$ more provided)
$$\frac{\Delta \vdash K \Longrightarrow K' \leq K'' \Longrightarrow K'''}{\Delta \vdash K \Longrightarrow K', \pi \bullet \tau \leq K'' \Longrightarrow K'''}$$

(Sub $\bullet$ less required)
$$\frac{\Delta \vdash K \Longrightarrow K' \leq K'' \Longrightarrow K'''}{\Delta \vdash K \Longrightarrow K' \leq K'', \pi \bullet \tau \Longrightarrow K'''}$$

(Sub $\circ$ more required)
$$\frac{\Delta \vdash K \Longrightarrow K' \leq K'' \Longrightarrow K'''}{\Delta \vdash K, \pi \circ \tau \Longrightarrow K' \leq K'' \Longrightarrow K'''}$$

(Sub $\circ$ less provided)
$$\frac{\Delta \vdash K \Longrightarrow K' \leq K'' \Longrightarrow K'''}{\Delta \vdash K \Longrightarrow K' \leq K'' \Longrightarrow K''', \pi \circ \tau}$$

(Sub $\bullet$ provided)
$$\frac{\Delta \vdash \tau \leq \tau'}{\Delta \vdash K \Longrightarrow K', \pi \bullet \tau \leq K \Longrightarrow K', \pi \bullet \tau'}$$

(Sub $\bullet$ required)
$$\frac{\Delta \vdash \tau' \leq \tau}{\Delta \vdash K, \pi \bullet \tau \Longrightarrow K' \leq K, \pi \bullet \tau' \Longrightarrow K'}$$

(Sub $\circ$ provided)
$$\frac{\Delta \vdash \tau' \leq \tau}{\Delta \vdash K \Longrightarrow K', \pi \circ \tau \leq K \Longrightarrow K', \pi \circ \tau'}$$

(Sub $\circ$ required)
$$\frac{\Delta \vdash \tau \leq \tau'}{\Delta \vdash K, \pi \circ \tau \Longrightarrow K' \leq K, \pi \circ \tau' \Longrightarrow K'}$$

Figure 6.16: Subtyping rules for configurator values.

the tag of the resource. The available resources in the set of required resources resource are contravariant (Rule (Sub $\bullet$ required)), and unsatisfied resources are covariant (Rule (Sub $\circ$ required)). On the set of provided resources, available resources are covariant (Rule (Sub $\bullet$ provided)), and unsatisfied resources are contravariant, (Rule (Sub $\circ$ provided)).

When accounting the introduction of provided and required ports, they are best if kept the same as we statically verify, in reconfiguration actions for instance, if port names do not clash with others (while internal elements may overlap others without disrupting the implementation).

Given such a subtyping relation on configurator types we now emphasise some details in the typing of expressions where subtyping of configurator types may have influence. For instance, unsatisfied resources may exist in the required resources of a configurator and still the configurator that requires them can form a well-formed component. The rule for the compose e expression in the type system for the language $\lambda_{\chi}^{\leq}$, which demands that the required resource sets are empty, can still type this component creation expression. The unsatisfied resources can be eliminated from the type by the subsumption rule (Rule (Val Subsumption)) and Rule .

(Val Compose) $\quad (K_{\circ} = \emptyset)$
$$\frac{\Delta \vdash e : \emptyset \Longrightarrow K}{\Delta \vdash \mathsf{compose}\ e : K_{\triangleleft} \Rightarrow K_{\triangleright}}$$

To type the composition of two configurators we also rely on subsumption to synchronise the types of the operands to be composed.

$$\text{(Comp Sequence)} \qquad\qquad (K'\#K'', K'\#K''')$$

$$\frac{\Delta \vdash e_1 : K \Longrightarrow K', K_c \quad \Delta \vdash e_2 : K_c, K'' \Longrightarrow K'''}{\Delta \vdash (e_1; e_2) : K, K'' \Longrightarrow K', K'''}$$

Notice that the additional available or unsatisfied resources in the types are naturally propagated to the resulting type. As for variances in the types of resources, subsumption can be used to make the common set of resources K_c to coincide.

Although this relation on configurator types is not integrated in the presented developments, we expect our current type safety results to hold in this extended subtyping relation.

6.5 An Example

We now illustrate the use of polymorphic values in our language by means of a small example. Consider that a public key distribution infrastructure is implemented by a component that has two internal components, and that these components are responsible for replicating keys in a network of servers. Each one of these components deals with a different kind of keys (characterised by two different subtypes of some type Key). Additionally, consider that although the distribution mechanism is the same for both kinds of keys, for some reason it is interesting to implement their replication separate. A polymorphic component KeyReplicator, of type TKeyReplicator, is therefore used twice to instantiate the internal components of our main component. Consider the following type declarations and the skeleton of a component using an update service especially designed for this kind of components. (We use in this example the Java-like syntax used in the examples shown in the Introduction.)

```
TKeyReplicator = All(X≤Key) {...}⇒{...}
IUpdateScript = All(X≤Key) {update:(TKeyReplicator<X>)void}
TUpdateScript = All(X≤Key) {}⇒{p:IUpdateScript<X>}
IReplicatorUpdateGateway = {availableScript:()boolean, getScript:()TUpdateScript}
...
Keychain = component {
  requires gw: IReplicatorUpdateGateway;
  ...
  uses replicator128 = KeyReplicator<Key128>;
  uses replicator64 = KeyReplicator<Key64>;
  ...
  methods m {
    checkAll() {
      if( gw.availableScript() ) {
        script = gw.getScript();

        (new (script<Key128>)).p.update(replicator128);
        (new (script<Key64>)).p.update(replicator64);
      }
    }
  };
  ...
}
```

Notice that the value returned by method getScript is a type abstraction which needs to be type instantiated to meet the type of the target object. The resulting component value can then be used to produce an object and its method update in port p can be called to safely upgrade both internal elements replicator128 and replicator64.

Now consider that the implementation of component KeyReplicator makes uses of a polymorphic container to hold the distributed keys. One reconfiguration action that makes the use of type parameters interesting is one that changes the implementation of such container. Consider the following component UpdateScript implementing such a reconfiguration action.

```
UpdateScript = All(X) component {
  provides p:IUpdateScript<X>;
  methods m {
    r = (uses x = CNewContainer<X>; plug x.p into c.q),
    void update(o:TKeyReplicator<X>) { reconfig r[o]; }
  };
  plug m into p
}
```

Consider that the container component inside of the target object (an instance of component KeyReplicator) is plugged to a port c.q of another internal component named c. Then, the new container (component CNewContainer) is introduced into the structure of the target object and its port p is plugged to port c.q to replace the old implementation of the container.

This example demonstrates the smooth integration of the component model with the polymorphic mechanisms just introduced. Notice that, by means of parametric polymorphism we are able to define reconfiguration actions at a high level of genericity.

6.6 ComponentJ

This section describes an experiment aim at the integration of the programming model presented in this work in a practical main-stream programming platform. To do so, we design a glue language, in a glue language called componentJ, based on our component calculus, $\lambda^{\leq}_{\chi}$, and targeting the Java programming and run-time environment. The language componentJ is an object-oriented imperative language where composition is preferred to implementation inheritance. Instead of classes and objects, the building blocks of componentJ applications are components and objects.

Unlike related approaches to the integration of component-based programming in object-oriented languages, such as Jiazzi [74], which is derived from the Unit's model [44], or more recently in Smart Modules [8], derived from the CMS calculus [11], we define components as first-class citizens, exactly as in $\lambda_{\chi}^{\leq}$, and component composition as a run-time operation, rather than a static linking procedure performed before run-time. Although componentJ includes both compositional and computational fragments, thus forming a general purpose language, we envisage it more as a glue and scripting language for componentJ and Java components. The componentJ language may be either used as the main language of a program, possibly controlling native Java components, or, the other way around, componentJ components can be used in Java programs.

Java was chosen as a template for the design of the scripting fragment of componentJ, and the Java environment as the target run-time environment not only due to its widespread use but also due to the presence of useful system level mechanisms such as dynamic loading, late binding, decoupling between types and implementation (interfaces and single inheritance), type safe reference semantics, and implicit garbage collection, all combined in a statically and strongly typed language. Another programming language based on $\lambda_{\chi}^{\leq}$, componentGlue was also developed in the context of [39], but targeting the .NET platform.

To adapt our programming model to a practical language, a set of constraints were imposed on the initial design. The most significant ones are related to typing. Unlike type equivalence (and subtyping) in $\lambda_{\chi}^{\leq}$, which is uniformly defined on the structure of types, componentJ subtyping for primitive types and port interfaces is based on type names. It turns out that a completely general structural subtype relation is hard to implement efficiently in the JVM because, by design, it adopts a nominal subtype relation based on an explicitly typed hierarchy of classes and interfaces. Nevertheless, when adding component types to the Java platform we preserved our intended design, relying on structural type equivalence to compare them.

We also have implemented a prototype componentJ compiler[1] which allows one to experiment with the language. Although the current compiler's implementation does not cover all features of our model, it allows us to play with a significant part of the language. Besides the examples provided in the component binary package, which illustrate the basic implemented features, the language was also tested in the context of a research project (DataBricks) [73], where, in collaboration with colleagues from the distributed systems area, it helped to modularly define a framework for data management in mobile applications.

The most significant features of the component calculus which have not been covered by the current compiler implementation are the reconfiguration of component instances and first-class composition operations.

A more detailed description of the componentJ system can be found in appendix B.

[1] This compiler can be downloaded from the author's web page: `http://www-ctp.di.fct.unl.pt/~jcs`.

6.7 Remarks

This concludes the presentation of our programming model featuring dynamic composition and reconfiguration of objects at the programming language level. In the preceding chapters, we have incrementally showed how these concepts can be abstracted in language constructs and how configurators and components can be freely combined to support the programming of software construction and software maintenance operations which are typical of the Component-Oriented Paradigm. We also showed how these constructs can be decorated with type annotations in order to statically ensure the absence of run-time errors.

In this chapter, we have presented the integration of our language with other standard programming language mechanisms such as inclusion and parametric polymorphism and recursive types. The uniform extension of the language with subtyping and bounded type abstraction allows for the manipulation of generic component and configurator values. Notice that the structure of component and configurator types is such that it also allows for the definition of structural subtyping relations based on services and resources.

It is interesting to observe the interaction of subsumption on configurators, components, and objects values with the composition and reconfiguration mechanisms. We know that, by subsumption, it may be the case that a component value has less required ports than the ones known at compile-time. If this happens when a component is introduced in a composition then there are plug operations whose target port does not exist. These are quite harmless plug operations that try to implement a service that is not actually needed. Notice that the subtyping relation on configurator types presented in 6.4 also introduces this kind of vacuous plug operations. Nevertheless, our type system ensures that all the essential connections are still performed.

Soundness of object structures is also preserved when subsumption and reconfiguration interact. We have showed that reconfiguration may change the interface of an object, it may add new provided ports to an object, which become accessible in the successful branch of the **reconfig** r[o] expression. The type system statically ensures that there are no conflicts between the port names which are known and the newly introduced ones. However, by subsumption, some ports of an object which may not be known at compile-time may be replaced causing an apparent conflict. Observe that reconfiguration works by applying the composition operations to the target instance, thus obtaining a modified copy of the original object value. It turns out that object values are only a facade for a network of elements that actually implements their behaviour. On reconfiguration actions, a new object value is produced which still leads to the object's network of elements. However, the old facade is preserved and therefore the apparently conflicting ports are kept in different contexts and all connections inside the instance are kept consistent, thus avoiding any kind of disruption.

We next conclude the dissertation by summarising the main results and emphasising the main characteristics of our programming model.

Chapter 7

Conclusions

In this dissertation we have presented a programming model we believe captures the essence of the Component-Oriented programming paradigm. We have introduced and justified a notion of software components as stateless entities which are capable of producing service-providing executable objects and that define the behaviour of such objects by aggregation and adaptation of other components. We instantiated our model in a core programming language where software management operations, typical of component-based systems, are abstracted by typeful programming language constructs.

Our model follows some basic design principles which state that all implicit dependencies between components should be made explicit at compile-time, that the construction of program structures should be controlled by run-time mechanisms, and that the evolution of software should be possible at the level of objects. An overall principle is that the operations introduced following the principles above should be verifiable at compile-time following a typeful programming language design.

We next enumerate the main characteristics of the model we propose.

The first aspect of our model that is worth noting is that the units of code reuse in our model are components, which are linked structures of components and scripting blocks, and configurators, which are composition operations. Both configurators and components are first-class values, that can be freely combined according to their type information using all the computational power of the base language. This allows the computation of all sorts of dynamic compositions of components and configurators, which may depend on run-time information and locally available elements.

Although the construction of configurator and component values depends on computations in a programming language with full computational power, and thus may not terminate, configurator values represent first-class programs in a language (that of the composition operations) which are executed in isolation with relation to the remaining computations. We ensure that all configuration programs terminate. As noted in the end of Section 3.1.1, this separation closely relates to the issue of phase distinction [22].

This separation between the definition of program structures and its actual realisation is also present in the nature of configurator and component values. Configurators and components are stateless values of the language that can be used to create and modify object structures at run-time. This is particularly useful in a distributed setting, where both components and configuration scripts may need to be stored and loaded from remote locations, or passed around in communication channels.

Another important characteristic of our model, is that composition operations form a declarative language (that describes the final structure of a component or object instead of describing how to achieve it) that uniformly expresses both the composition of configurators and components, and the reconfiguration of objects. We have defined a small set of elementary composition operations (perhaps minimal) that may be incrementally combined to obtain more elaborate composition operations. Each elementary operation produces a distinguished effect on defining a component or reconfiguring an object.

The consistency of component compositions, and hence of object structures, is statically enforced by a type system. Although defined from rather standard language constructs (a λ-calculus with mutable records), our language does not seem straightforwardly encodable, in a type preserving way, in such a canonical language, due to the presence of intensional information at the level of types, which assigns to configurator values type information not related to their behaviour (during computation) but instead denotes the effect they produce on object structures.

Due to information hiding on objects and components, type safety of reconfiguration actions on objects results from a combination of static type checking with a localised and efficient dynamic type check. Run-time type information is kept in objects and configurators to support this test. Furthermore, configurator composition requires computation with type information to take place. An analogy can be made to some approaches to mobile code and dynamic loading of code that use hashing or signature information. Our model has the additional advantage that we obtain this information by direct composition of the information in configurator values instead of (re)processing the source that generated them. Since we can (re)check the type information in the configurator values against the operations they contain, this mechanism can also be seen as a simple form of proof carrying code [75].

The last addition to our model showed that subtyping and bounded parametric polymorphism work uniformly with components and configurators thus allowing for generic components and configurators to be transmitted in communication channels. Structural subtyping relations on component types are based on the declared services they require or provide, thus defining a flexible way of reusing components. Although not formally integrated in the model, we also use intensional type information to sketch a subtyping relation on configurator types.

In addition to the component-based programming model, we have developed new techniques that apply to the subtyping problem of second-order equi-recursive types. We developed them on a standard and canonical second-order λ-calculus, kernel-Fun, and then applied them to our component language. We uniformly extended a standard approximation to first-order equi-recursive types with explicit management of type variables and a similarity relation on subtyping judgements. The result is a coinductive definition of the subtyping relation and the corresponding subtyping algorithm which is significantly simpler than other existing approaches. In particular, our approach extends a standard coinductive algorithm with relevant differences in its halting condition where we consider our similarity relation instead of standard syntactic equivalence.

All the characteristics mentioned above are formally defined in the programming languages presented in this dissertation. Big step operational semantics and type systems are defined and combined in order to prove type safety for these constructions. An initial draft of our model was published in [81] and the more important results were formally presented in [87, 88].

The work enclosed in this dissertation is by no means complete, many topics remain unstudied, either because they represent branching matters towards other areas, like the integration with distribution and concurrency in our model [39], or they represent an undesired widening of the focus we imposed to ourselves, such as the issue of state transformation on reconfiguration actions. We next describe some possible directions where to use and expand the obtained results.

At the level of the component calculus, it is conceivable, in principle, to extend the application of reconfiguration not only to objects, but also to component values. We refrained from pursuing that because it does not seem to increase the expressiveness of our language, and lacks pragmatical motivation. Nevertheless, this points to possible developments of a component versioning mechanism, at the level of the supporting platform, which would centralise the evolution of components and consequently of all their instances. Persistency of these changes is also an issue that can be studied in this setting. Some more developments are also needed, at the level of a practical supporting framework, for instance, to implement the dynamic loading of typed component values.

We modelled dynamic reconfiguration of instances based on the internal structure of objects, regardless of their state. The evolution of state is not treated in our proposal, and can be explicitly expressed in the base language. However, it would be interesting to investigate safe ways of evolving the state of objects in the context of a reconfiguration in our model. Namely, an embryo was already put into practise in the context of componentJ prototype compiler. The idea consists on freezing the state of an object and producing a component out of it. This component can then produce instances whose initial state is the frozen state of the original instance. The object's configuration, given by the connections to its required ports, is undone. Thus, the new component must be configured again in the new instantiation context.

In terms of the fundamental results on subtyping of recursive second-order types, it would also be interesting to investigate more flexible typing relations, involving subtyping, polymorphism, and recursive types, along the lines of [87] to which our results apply. Namely, the ones that fit better the object-oriented style. For instance, in usual class-based languages object types are usually extended by adding more methods rather than refining the signatures of existing ones. A subtyping relation that relates the bounds of type parameters in this width-only subtyping discipline fits the necessary conditions to define a correct subtyping algorithm. A precise characterisation of these relations remains to be done.

Another unexplored aspect relates to the representation of abstract data types in our model. As in any object-oriented language, binary methods based on interface types hide the internal details of objects passed as arguments. This hinders the definition of binary relations on values with the same implementation, which is quite natural, in class-based languages, by using class names, which are types and references to a particular implementation. Our model could be orthogonally extended with pure abstract data types by means of bounded existential types exported by components. This would surely involve, in the case of reconfiguration, the usage of versioning techniques like the ones described in [90, 37], to tackle conversions between different versions of abstractly typed values.

From our point of view, the present work surely represents a firm step into the construction of programming language mechanisms that capture the essence of the Component-Oriented programming paradigm. The composition of independently developed and possibly distributed executing services yields the need for increasing support, not only at the binary level of binding, but also at the programming language level, with proper abstraction and verification mechanisms. This work establishes ground for basic type checking verification of software composition and leaves an open track to investigate the usage of more sophisticated forms of typing for services.

Appendix A

Complete proofs

A.1 Chapter 2

In this section we give the detailed proofs of the lemmas in chapter 3.
(This is a repetition of Theorem 2.24, defined in page 37.)

Theorem 2.24 (Subject Reduction). *Let $(e; S)$ be a valid configuration in $\lambda_R^\tau \backslash \{\mathsf{nil}\}$ and let Γ be a typing environment typing S and $\mathsf{nil}(S) = \emptyset$:*
If $\Gamma \vdash e : \tau$ and $e; S \downarrow v; S'$ then:

a) there is a Γ' that extends Γ and types S',

b) $\Gamma' \vdash v : \tau$,

c) v is either an abstraction, or a location that is either `undefined` or `refers-to` a record, and

d) such that $\mathsf{nil}(S') = \emptyset$.

Proof. We prove this theorem by induction on the height of the evaluation derivation and in the cases of the last rule used, we show in the cases that evaluate to wrong that these rules are never applicable.

Case: **(Eval Value)**

If v is a location, then it must be correctly typed by Γ and therefore the lemma holds by definition 2.21. If v is a record then its locations are correctly typed in Γ and the lemma also holds. If v is an abstraction the conclusions of the theorem hold with $\Gamma' = \Gamma$, $S' = S$, and $v = \lambda x : \tau.e$. We also have that $\mathsf{nil}(S') = \emptyset$.

Case: **(Eval Application)$^\tau$**

In this case, with $e_1(e_2); S \downarrow v; S'$, we know by Rule (Eval Application), that: a) $e_1; S \downarrow \lambda x : \tau.e; S'$,

b) $e_2; S' \downarrow v_2; S''$, and c) $e[x \leftarrow v_2]; S'' \downarrow v; S'''$. On the typing derivation, the only plausible last rule for $\Gamma \vdash e_1(e_2) : \sigma$ is Rule (Val Application), with the premises: d) $\Gamma \vdash e_1 : \tau \rightarrow \sigma$ and e) $\Gamma \vdash e_2 : \tau$.

Taking d) and a), by induction hypothesis we conclude that there is a Γ' extending Γ and typing S', such that f) $\Gamma' \vdash \lambda x : \tau . e : \tau \rightarrow \sigma$. By Lemma 3.24 (weakening) on e) we conclude that $\Gamma' \vdash e_2 : \tau$, which together with b), by induction hypothesis, there is a Γ'' extending both Γ' and Γ and typing S'' such that g) $\Gamma'' \vdash v_2 : \tau$ and $\mathsf{nil}(S'') = \emptyset$. From f), by Lemma 3.24 (weakening) and Rule (Val Abstraction) we have that $\Gamma'', x : \tau \vdash e : \sigma$ and by Lemma 3.28 (substitution) with g) we have that h) $\Gamma'' \vdash e[x \leftarrow v_2] : \sigma$. Taking h) and c), by induction hypothesis we have that there is Γ''' extending Γ'' and typing S''', such that v is typed $\Gamma''' \vdash v : \sigma$ and v is either an abstraction or a location that is either *undefined* or *refers-to* a record, and $\mathsf{nil}(S''') = \emptyset$.

Case: **(Eval Record)**

When the last rule is (Eval Record) we have the hypothesis $[\ell_i = e_i \ ^{i \in 1..n}]; S_0 \downarrow \iota; S'$ with $S' = S_n[\iota_i \mapsto v_i \ ^{i \in 1..n}][\iota \mapsto \{\ell_i = \iota_i \ ^{i \in 1..n}\}]$ and $\Gamma \vdash [\ell_i = e_i \ ^{i \in 1..n}] : \{\!|\ell_i : \tau_i \ ^{i \in 1..n}|\!\}$. By the Rules (Val Record) and (Eval Record), we have that a) $\Gamma \vdash e_i : \tau_i$ and b) $e_i; S_{i-1} \downarrow v_i; S_i$ with $i \in 1..n$. By iterating the induction hypothesis with a) and b), and using Lemma 3.24 (weakening) to adjust the typing environment in the hypothesis, we reach the conclusion that, for all $i \in 1..n$, there is a Γ_i extending Γ_{i-1} and typing each S_i, with $\Gamma_0 = \Gamma$ such that $\Gamma_i \vdash v_i : \tau_i$ and $\mathsf{nil}(S_i) = \emptyset$. So, by transitivity, we have that Γ_n types S_n and extends Γ with $\mathsf{nil}(S_n) = \emptyset$. By Rule (Val Record Value) and Definition 2.21 we have that $\Gamma' = \Gamma_n, \iota : \{\!|\ell_i : \tau_i \ ^{i \in 1..n}|\!\}$ types S' and that $\Gamma' \vdash \iota : \{\!|\ell_i : \tau_i \ ^{i \in 1..n}|\!\}$ with relation to S'. We have that ι is a location that *refers-to* a record and $\mathsf{nil}(S') = \emptyset$.

Case: **(Eval Assign)**

In this case, from $e_1.\ell := e_2$, by Rule (Eval Assign), we have that a) $e_1; S \downarrow \iota; S'$ with $\iota' = \mathrm{deref}_{S'}(\iota)$ and $S'(\iota') = \{\ldots, \ell = \iota'', \ldots\}$, and b) $e_2; S' \downarrow v; S''$. By Rule (Val Assign), we have that $\Gamma \vdash (e_1.\ell := e_2) : \sigma$ is supported by c) $\Gamma \vdash e_1 : \{\!| \ldots, \ell : \sigma, \ldots |\!\}$ and d) $\Gamma \vdash e_2 : \sigma$. Thus, from a) and c), by induction hypothesis we know that there is a Γ' extending Γ and typing S' such that $\Gamma' \vdash \iota : \{\!| \ldots, \ell : \sigma, \ldots |\!\}$ and $\mathsf{nil}(S') = \emptyset$. By Definition 2.21 we conclude that for some $\iota' = \mathrm{deref}_{S'}(\iota)$ we have $S'(\iota') = \{\ldots, \ell = \iota'', \ldots\}$ and that $\Gamma \vdash S'(\mathrm{deref}_{S'}(\iota)) : \{\!| \ldots, \ell : \sigma, \ldots |\!\}$. By rule (Val Record) we conclude that e) $\Gamma' \vdash \iota'' : \sigma$. From b), by Lemma 3.24 (weakening) we obtain $\Gamma' \vdash e_2 : \sigma$, and by induction hypothesis together with d) we have that there is a Γ'' extending Γ' and typing S'' such that f) $\Gamma'' \vdash v : \sigma$. and $\mathsf{nil}(S'') = \emptyset$. Γ' types $S''[\iota \mapsto v]$ and the result of the assignment expression is v, and v is not nil we conclude this case with $\mathsf{nil}(S''[\iota \mapsto v]) = \emptyset$. with v satisfying the conclusions of the theorem.

Case: **(Eval Select)**

If the last rule used is (Eval Select) then we have that a) $e_1; S \downarrow \iota; S'$ with $\iota' = \mathrm{deref}_{S'}(\iota)$ and

$S'(\iota') = \{\ldots, \ell = \iota'', \ldots\}$. On the typing relation we have that c) $\Gamma \vdash e : \{\!|\, \ldots, \ell : \tau, \ldots \,|\!\}$. From a) and c), by induction hypothesis we know that there is a Γ' extending Γ and typing S' such that $\Gamma' \vdash \iota : \{\!|\, \ldots, \ell : \sigma, \ldots \,|\!\}$ and $\mathsf{nil}(S') = \emptyset$. By Definition 2.21 we conclude that for some $\iota' = \mathrm{deref}_{S'}(\iota)$ we have $S'(\iota') = \{\ldots, \ell = \iota'', \ldots\}$ and that $\Gamma \vdash S'(\iota') : \{\!|\, \ldots, \ell : \sigma, \ldots \,|\!\}$. By Rule (Val Record) we conclude that e) $\Gamma' \vdash \iota'' : \sigma$. By Definition 2.21 we conclude that $\Gamma' \vdash S'(\iota'') : \sigma$. Notice that $\mathsf{nil}(S') = \emptyset$.

We now look into the cases where the evaluation yields wrong and show that these rules are not applicable to well-typed expressions.

***Case:* (Wrong Call)**

For a application to be well-typed we must have $\Gamma \vdash e_1(e_2) : \sigma$. Then, by inspection of the type system, we conclude that the only typing rule that may derive this judgment is Rule (Val Application), and therefore we have a) $\Gamma \vdash e_1 : \tau \rightarrow \sigma$ and b) $\Gamma \vdash e_2 : \sigma$. We also have that c) $e_1; S \downarrow v; S'$ which by induction hypothesis lets us conclude that there is a Γ' typing S' such that $\Gamma \vdash v : \tau \rightarrow \sigma$ and v is an abstraction. Therefore if an application expression is well-typed, the Rule (Wrong Call) is never applicable.

***Case:* (Wrong Assign)**

If $\Gamma \vdash e_1.\ell := e_2 : \tau$ then by Rule (Eval Assign) we know that a) $\Gamma \vdash e_1 : \{\!|\, \ldots, \ell : \tau, \ldots \,|\!\}$. The evaluation hypothesis has the premise b) $e_1; S \downarrow v; S'$. Taking a) and b), by induction hypothesis we know that there is Γ' typing S' and extending Γ such that $\Gamma \vdash v : \{\!|\, \ldots, \ell : \tau, \ldots \,|\!\}$ and v must be a location leading to a record. Thus Rule (Wrong Assign) is never applicable to well-typed assignment expressions.

***Case:* (Wrong Assign 2)**

Following the reasoning from the previous case, we have $\Gamma \vdash \iota : \{\!|\, \ldots, \ell : \tau, \ldots \,|\!\}$ and that ι is a location leading to a record. By Definition 2.21 we know that $\Gamma \vdash S(\iota) : \{\!|\, \ldots, \ell : \tau, \ldots \,|\!\}$ which by Rule (Val Record) must contain the label ℓ and therefore Rule (Wrong Assign 2) is also never applicable.

***Case:* (Wrong Select)**. Similar.

***Case:* (Wrong Select 2)**. Similar.

In this way we conclude that all evaluation derivations on well-typed expressions must contain only rules of Figure 2.8 and therefore the subject reduction property holds in the operational semantics defined in Definition 2.16 □

(This is a repetition of Lemma 3.29, defined in page 69.)

Lemma 3.29 (Subject Reduction).

1. *Let $(e;S)$ be a valid configuration in $\lambda^{\tau}_{\chi}\backslash\{\mathsf{nil}\}$ such that $\mathsf{nil}(S) = \emptyset$ and let Γ be a typing environment typing S such that e and S are record-based with relation to Γ.*

   If $\Gamma \vdash e:\tau$ and $(e;S \downarrow v;S')$ then:

   a) *there is a Γ' that extends Γ and types S',*

   b) *$\Gamma' \vdash v:\tau$,*

   c) *v is either an abstraction, a configurator, a component, or a location that is either `undefined` or `refers-to` a record,*

   d) *v and S' are record-based with relation to Γ, and*

   e) *$\mathsf{nil}(S') = \emptyset$.*

2. *Let c be an expression such that $\Gamma \vdash c:K \Longrightarrow K'$, let S be a heap such that, for some set $X \in Dom(S)$, $\mathsf{nil}(S) \subseteq \{\mathrm{select}_S(s,\pi) \mid (\pi:\tau) \in K_{\lhd} \cup K_{\circ}\} \uplus X$ and Γ types S and S is record-based with relation to Γ.*

   Let s be a partially linked object s such that it conforms with K and its partially linked object type is $[\![R \oplus K_{\lhd} \Rightarrow P \oplus K_{\rhd}]\!]$ and s and S are record-based with relation to Γ:

   If $(r,e,p);c;S \Downarrow (r',e',p');S'$ then

   a) *there is Γ' typing S' and extending Γ,*

   b) *$s' = (r',e',p')$ is a partially linked object that extends s and conforms with K'. Its partially linked object type is $[\![R, K'_{\lhd} \Rightarrow P, K'_{\rhd}]\!]$, and*

   c) *s' and S' are record-based with relation to Γ', and*

   d) *$\mathsf{nil}(S') \subseteq \{\mathrm{select}_{S'}(s,\pi) \mid (\pi:\tau) \in K'_{\lhd} \cup K'_{\circ}\} \uplus X$.*

Proof. The proof is carried out by induction on the two cases of the lemma. On the first case, we prove it by induction on the height of the evaluation derivation and in the cases of the last rule used. We use the second case when needed and its conditions are met. We show, in the cases that evaluate to wrong, that these rules are never applicable.

***Case:* (Eval Value)**

The conclusions of the theorem hold with $\Gamma' = \Gamma$, $S' = S$, and $\Gamma' \vdash v:\tau$. We also have that $\mathsf{nil}(S') = \emptyset$.

***Case:* (Eval Application)$^{\tau}$**

In this case, with $e_1(e_2);S \downarrow v;S'$, we know by Rule (Eval Application), that: a) $e_1;S \downarrow \lambda x:\tau.e;S'$, b) $e_2;S' \downarrow v_2;S''$, and c) $e[x \leftarrow v_2];S'' \downarrow v;S'''$. On the typing derivation, the only plausible last rule for $\Gamma \vdash e_1(e_2) : \sigma$ is Rule (Val Application), with the premises: d) $\Gamma \vdash e_1 : \tau \rightarrow \sigma$ and e) $\Gamma \vdash e_2 : \tau$.

Taking d) and a), by induction hypothesis we conclude that there is a Γ' extending Γ and typing S', such that f) $\Gamma' \vdash \lambda x : \tau . e : \tau \rightarrow \sigma$. By Lemma 3.24 (weakening) on e) we conclude that $\Gamma' \vdash e_2 : \tau$, which together with b), by induction hypothesis, implies that there is a Γ'' extending both Γ' and Γ and typing S'' such that g) $\Gamma'' \vdash v_2 : \tau$ and $\mathsf{nil}(S'') = \emptyset$. From f), by Lemma 3.24 (weakening) and Rule (Val Abstraction) we have that $\Gamma'', x : \tau \vdash e : \sigma$ and by Lemma 3.28 (substitution) with g) we have that h) $\Gamma'' \vdash e[x \leftarrow v_2] : \sigma$.

Taking h) and c), by induction hypothesis we have that there is Γ''' extending Γ'' and typing S''', such that v is typed $\Gamma''' \vdash v : \sigma$ and v is either an abstraction, a configurator, a component, or a location that is either *undefined* or *refers-to* a record, and $\mathsf{nil}(S''') = \emptyset$.

The application of induction hypothesis to the different premises and the application of Lemma 3.28 ensures that v and S''' are record-based.

***Case:* (Eval Record)**

When the last rule is (Eval Record) we have the hypothesis $[\ell_i = e_i{}^{i\in 1..n}]; S_0 \downarrow \iota; S'$ with $S' = S_n[\iota_i \mapsto v_i{}^{i\in 1..n}][\iota \mapsto \{\ell_i = \iota_i{}^{i\in 1..n}\}]$ and $\Gamma \vdash [\ell_i = e_i{}^{i\in 1..n}] : \{\!|\ell_i : \tau_i{}^{i\in 1..n}|\!\}$. By Rules (Val Record) and (Eval Record), we have that a) $\Gamma \vdash e_i : \tau_i$ and b) $e_i; S_{i-1} \downarrow v_i; S_i$ with $i \in 1..n$. By iterating the induction hypothesis with a) and b), and using Lemma 3.24 (weakening) to adjust the typing environment in the hypothesis, we reach the conclusion that, for all $i \in 1..n$, there is a Γ_i extending Γ_{i-1} and typing each S_i, with $\Gamma_0 = \Gamma$ such that $\Gamma_i \vdash v_i : \tau_i$ and $\mathsf{nil}(S_i) = \emptyset$. So, by transitivity, we have that Γ_n types S_n and extends Γ with $\mathsf{nil}(S_n) = \emptyset$. By Rule (Val Record Value) and Definition 2.21 we have that $\Gamma' = \Gamma_n, \iota : \{\!|\ell_i : \tau_i{}^{i\in 1..n}|\!\}$ types S' and that $\Gamma' \vdash \iota : \{\!|\ell_i : \tau_i{}^{i\in 1..n}|\!\}$ with relation to S'. We have that ι is a location that *refers-to* a record and $\mathsf{nil}(S') = \emptyset$. The result is a location typed by a record type and therefore it is record-based. The resulting heap is also record-based since all the introduced values are record-based (by induction hypothesis).

***Case:* (Eval Assign)**

In this case, from $e_1.\ell := e_2$, by Rule (Eval Assign), we have that a) $e_1; S \downarrow \iota; S'$ with $\iota' = \mathrm{deref}_{S'}(\iota)$ and $S'(\iota') = \{\ldots, \ell = \iota'', \ldots\}$, and b) $e_2; S' \downarrow v; S''$. By Rule (Val Assign), we have that $\Gamma \vdash (e_1.\ell := e_2) : \sigma$ is supported by c) $\Gamma \vdash e_1 : \{\!|\ldots, \ell : \sigma, \ldots|\!\}$ and d) $\Gamma \vdash e_2 : \sigma$. Thus, from a) and c), by induction hypothesis we know that there is a Γ' extending Γ and typing S' such that $\Gamma' \vdash \iota : \{\!|\ldots, \ell : \sigma, \ldots|\!\}$. and $\mathsf{nil}(S') = \emptyset$. By Definition 2.21 we conclude that $S'(\mathrm{deref}_{S'}(\iota)) = \{\ldots, \ell = \iota'', \ldots\}$ and that $\Gamma \vdash S'(\mathrm{deref}_{S'}(\iota)) : \{\!|\ldots, \ell : \sigma, \ldots|\!\}$. By rule (Val Record) we conclude that e) $\Gamma' \vdash \iota'' : \sigma$. From b), by Lemma 3.24 (weakening) we obtain $\Gamma' \vdash e_2 : \sigma$, and by induction hypothesis together with d) we have that there is a Γ'' extending Γ' and typing S'' such that f) $\Gamma'' \vdash v : \sigma$. and $\mathsf{nil}(S'') = \emptyset$. Γ' types $S''[\iota \mapsto v]$ and the result of the assignment expression is v, and v is not nil we conclude this case with $\mathsf{nil}(S''[\iota \mapsto v]) = \emptyset$.

with v satisfying the conclusions of the theorem. Value v is record-based and it is introduced inside a record, so the resulting heap $S''[\iota \mapsto v]$ is also record-based.

Case: **(Eval Select)**

If the last rule used is (Eval Select) then the expression $e.\ell$ can be typed by two possible rules. In both cases we have that a) $e_1; S \downarrow \iota; S'$ with $\iota' = \mathrm{deref}_{S'}(\iota)$ and $S'(\iota') = \{\dots, \ell = \iota'', \dots\}$. The subcases are:

Subcase: **(Val Select)**

Now, in this case we have that c) $\Gamma \vdash e : \{\!|\, \dots, \ell : \tau, \dots \,|\!\}$. From a) and c), by induction hypothesis we know that there is a Γ' extending Γ and typing S' such that $\Gamma' \vdash \iota : \{\!|\, \dots, \ell : \sigma, \dots \,|\!\}$ and $\mathsf{nil}(S') = \emptyset$. By Definition 2.21 we conclude that for some $\iota' = \mathrm{deref}_{S'}(\iota)$ we have $S'(\iota') = \{\dots, \ell = \iota'', \dots\}$ and that $\Gamma \vdash S'(\iota') : \{\!|\, \dots, \ell : \sigma, \dots \,|\!\}$. By Rule (Val Record) we conclude that e) $\Gamma' \vdash \iota'' : \sigma$. By Definition 2.21 we conclude that $\Gamma' \vdash S'(\iota'') : \sigma$. Notice that $\mathsf{nil}(S') = \emptyset$. The resulting value results from consulting a record in a record-based heap, so, it is also record-based.

Subcase: **(Val Select Interface)**

In this case we know that c) $\Gamma \vdash e : \{\dots, \ell : \tau, \dots\}$ and therefore, by induction hypothesis (from a) and c)), that e_1 yields an interface typed value (ι). By inspection of the type system (Definition 3.18) we see, by Rule (Val Select Interface), that $\Gamma \vdash \iota : \{\!|\, \dots, \ell : \tau, \dots \,|\!\}$ and then the reasoning follows by applying Definition 2.21 and Rule (Val Record) as in the subcase above.

Case: **(Eval Compose)**

We have the typing $\Gamma \vdash \mathsf{compose}\ e : K_{\lhd} \Rightarrow K_{\rhd}$ supported by $\Gamma \vdash e : \emptyset \Longrightarrow K$ where $K_{\circ} = \emptyset$. The evaluation hypothesis is $\mathsf{compose}\ e; S \downarrow \mathsf{comp}(c); S'$ where $e; S \downarrow \mathsf{conf}(c); S'$. By induction hypothesis we have that there is a Γ' extending Γ, that types S' and $\Gamma' \vdash \mathsf{conf}(c) : \emptyset \Longrightarrow K$ and $\mathsf{nil}(S') = \emptyset$. By Rule (Val Configurator Value) we know that $\Gamma' \vdash c : \emptyset \Longrightarrow K$, and by Rules (Val Compose) and (Val Composition Value) we conclude that $\Gamma' \vdash \mathsf{comp}(c) : K_{\lhd} \Rightarrow K_{\rhd}$. with Γ' extending Γ and typing S', and with the resulting value being a component value. From the application of the induction hypothesis we also conclude that $\mathsf{nil}(S') = \emptyset$ and that the value $comp(c)$ is record-based.

Case: **(Eval New)**

The hypothesis $\Gamma \vdash \mathsf{new}\ e\ \mathsf{with}\ \ell^r_j := e_j\ {}^{j \in 1..n} : \{\ell^p_i : \sigma_i\ {}^{i \in 1..n}\}$ is obtained by Rule (Val New) with the premises $\Gamma \vdash e : \tau$ with $\tau = \{\ell^r_j : \tau_j\ {}^{j \in 1..m}\} \Rightarrow \{\ell^p_i : \sigma_i\ {}^{i \in 1..n}\}$. new is evaluated by Rule (Eval New), thus, if $\mathsf{new}\ e\ \mathsf{with}\ \ell^r_j := e_j\ {}^{j \in 1..m}; S \downarrow \iota; S'$, then we must have $e; S \downarrow \mathsf{comp}(c); S_0$. By induction hypothesis we have that there is a Γ_0 extending Γ such that $\Gamma_0 \vdash \mathsf{comp}(c) : \tau$ and $\mathsf{nil}(S_0) = \emptyset$.

Now, notice that then presentation order of the premises in the rule does not correspond to the real dependence between them. For technical reasons, the plug-assignments must be considered first in this proof. By iteratively applying Lemma 3.28 (substitution), the induction hypothesis, and Lemma 3.24 (weakening) on the typing and evaluation judgements of each plug-assignment expression, $(\Gamma \vdash e_i : \tau_i)$ and $(e_i; S_{i-1} \downarrow v_i; S_i)$, we obtain that for all $i \in 1..n$ we have Γ_i extending Γ_{i-1} and typing S_i such that $\Gamma_i \vdash v_i : \tau_i$ and $\mathsf{nil}(S_i) = \emptyset$. Notice that all v_i and S_i are record-based.

By Lemma 3.24 (weakening) on the typing of $\mathsf{comp}(c)$ and Rule (Val Composition Value) we have $\Gamma_n \vdash c : \emptyset \Longrightarrow K$ and $K_\circ = \emptyset$, with type $\tau = K_\triangleleft \Rightarrow K_\triangleright$ thus $K_\triangleleft = \{\ell_j : \tau_j\ ^{j \in 1..m}\}$. Notice that the judgement above applies c to the empty instance $(\mathbf{0}; c; S_n \Downarrow s; S_{n+1})$, and that $\mathsf{nil}(S_n) = \emptyset$ meets the conditions of the second part of the lemma with $X = \emptyset$. In this case, $\mathbf{0}$ conforms with the resource list $\emptyset$ and has type $[\![\{\} \Rightarrow \{\}]\!]$. By induction hypothesis on the second case of the lemma, we have that there is a Γ_{n+1} extending Γ_n and the resulting instance s conforms with K with the partial type $[\![K_\triangleleft \Rightarrow K_\triangleright]\!]$. We know that the locations of the required ports are $\{\, l_i\ ^{i \in 1..n}\} = \{\mathrm{select}_{S'}(s, \ell) \mid (\ell : \tau) \in K_\triangleleft\}$ and that $\mathsf{nil}(S') \subseteq \{\mathrm{select}_{S'}(s, \ell) \mid (\ell : \tau) \in K_\triangleleft\}$.

Γ_{n+1} types S_{n+1} and the locations in $s_\triangleleft = \{\ell_i = l_i\ ^{i \in 1..n}\}$ are in S_n. From the conformance of s with K, Definition 3.23, and Rule (Val Record) we know that $\Gamma_{n+1} \vdash s : \{\!| \ldots, \ell_j^p : \sigma_j\ ^{j \in 1..m} |\!\}$ where $K_\triangleright = \{\ell_j^p : \sigma_j\ ^{j \in 1..m}\}$ and by Rule (Val Interface), we have a $\Gamma' = \Gamma_{n+1}, l : K_\triangleright$, extending Γ, and typing $S_{n+1}[\, l \mapsto s][\, l_i {\mapsto} v_i\ ^{i \in 1..n}]$. We finally have that l is a location that ***refers-to*** an object, which is a record, such that $\Gamma' \vdash\ l : K_\triangleright$ with $K_\triangleright = \{\ell_j^p : \sigma_j\ ^{j \in 1..m}\}$. Finally, since $v_i \neq \mathsf{nil}$ for all $i \in 1..n$ we have that $\mathsf{nil}(S_{n+1}[\, l \mapsto s][\, l_j {\mapsto} v_j\ ^{j \in 1..m}]) = \emptyset$ with s being the newly created instance. Notice that l and $S_{n+1}[\, l \mapsto s][\, l_j {\mapsto} v_j\ ^{j \in 1..m}]$ are record-based.

***Case:* (Eval Requires)**

The typing hypothesis for this case is $\Gamma \vdash (\mathsf{requires}\ \ell : I) : \sigma$ with $\sigma = \emptyset \Longrightarrow \{\ell \bullet I, \ell \triangleleft I\}$ where the expression $\mathsf{requires}\ \ell : \tau$ evaluates to a configurator. By Rule (Val Configurator Value), we have $\Gamma' \vdash \mathsf{conf}(\mathsf{requires}\ \ell : I) : \sigma$ with $\Gamma' = \Gamma$ and $S' = S$ with $\mathsf{nil}(S') = \emptyset$.

***Case:* (Eval Provides)**

The typing hypothesis for this case is $\Gamma \vdash (\mathsf{provides}\ \ell : \tau) : \sigma$ with $\sigma = \emptyset \Longrightarrow \{\ell \circ \tau, \ell \triangleright \tau\}$ where the expression $\mathsf{requires}\ \ell : \tau$ evaluates to a configurator. By Rule (Val Configurator Value), we have $\Gamma \vdash \mathsf{conf}(\mathsf{provides}\ \ell : \tau) : \sigma$ with $\Gamma' = \Gamma$ and $S' = S$ with $\mathsf{nil}(S') = \emptyset$.

***Case:* (Eval Plug)**

The typing hypothesis for this case is $\Gamma \vdash \mathsf{plug}\ (\pi_1 : \tau)\ \mathsf{into}\ (\pi_2 : \tau) : \sigma$ with
$\sigma = (\{\pi_2 \circ \tau, \pi_1 \bullet \tau\} \Longrightarrow \{\pi_1 \bullet \tau\})$ where the expression $\mathsf{plug}\ (\pi_1 : \tau)\ \mathsf{into}\ (\pi_2 : \tau)$ evaluates to a configurator. By Rule (Val Configurator Value), we have $\Gamma \vdash \mathsf{conf}(\mathsf{plug}\ (\pi_1 : \tau)\ \mathsf{into}\ (\pi_2 : \tau)) : \sigma$ with $\Gamma' = \Gamma$ and $S' = S$ with $\mathsf{nil}(S') = \emptyset$.

Case: **(Eval Method Block)**

The typing hypothesis for this case is $\Gamma \vdash x_K[\ell_i : \tau_i = e_i \ ^{i \in 1..n}] : \sigma$ with $\sigma = K \Longrightarrow K, \{x \bullet \{\ell_i : \tau_i \ ^{i \in 1..n}\}\}$ where the expression $x_K[\ell_i : \tau_i = e_i \ ^{i \in 1..n}]$ evaluates to a configurator. By Rule (Val Configurator Value), we have $\Gamma \vdash \mathsf{conf}(x_K[\ell_i : \tau_i = e_i \ ^{i \in 1..n}]) : \sigma$ with $\Gamma' = \Gamma$ and $S' = S$ with $\mathsf{nil}(S') = \emptyset$.

Case: **(Eval Uses)**

The evaluation of this expression produces a configurator where e is replaced by its resulting value, $x[e : \tau \Rightarrow \sigma]; S \downarrow \mathsf{conf}(x[v : \tau \Rightarrow \sigma]); S'$. The typing judgment implies $\Gamma \vdash x[e : \tau \Rightarrow \sigma] : \emptyset \Longrightarrow K$ with $\tau = \{\ell_i^r : \tau_i \ ^{i \in 1..n}\}, \sigma = \{\ell_j^p : \sigma_j \ ^{j \in 1..m}\}$ and $K = \{x \bullet \sigma, x.\ell_i^r \circ \tau_i \ ^{i \in 1..n}, x.\ell_j^p \bullet \sigma_j \ ^{j \in 1..m}\}$.

We have that $\Gamma \vdash e : \tau \Rightarrow \sigma$ and $e; S \downarrow v; S'$, which by induction hypothesis on the first part of the lemma implies that there is a Γ' extending Γ such that $\Gamma' \vdash v : \tau \Rightarrow \sigma$ and $\mathsf{nil}(S') = \emptyset$. The resulting value and the resulting heap are record-based by the application of the induction hypothesis.

We conclude, by Rule (Val Configurator Value), that $\Gamma' \vdash \mathsf{conf}(x[v : \tau]) : \emptyset \Longrightarrow K$ and therefore there is Γ' extending Γ and $\Gamma' \vdash \mathsf{conf}(x[v : \tau]) : \emptyset \Longrightarrow K$.

Case: **(Eval Sequence)**

By induction hypothesis we reach the conclusion that there is a Γ' extending Γ and typing S' such that $\Gamma' \vdash \mathsf{conf}(c_1) : K \Longrightarrow K', K_c$ and $\mathsf{nil}(S') = \emptyset$. Also by induction hypothesis, there is Γ'' extending both Γ' and Γ and typing S'' such that $\Gamma'' \vdash \mathsf{conf}(c_2) : K_c, K'' \Longrightarrow K'''$ and $\mathsf{nil}(S'') = \emptyset$. By Rule (Val Configurator Value) we conclude that $\Gamma'' \vdash \mathsf{conf}(c_1; c_2) : K, K'' \Longrightarrow K', K'''$. The resulting value and the resulting heap are record-based by the application of the induction hypothesis.

We follow by proving that well-typed expressions never evaluate to wrong.

Case: **(Wrong Call)**

For a application to be well-typed we must have $\Gamma \vdash e_1(e_2) : \sigma$. Then, by inspection of the type system, we conclude that the only typing rule that may derive this judgment is Rule (Val Application), and therefore we have a) $\Gamma \vdash e_1 : \tau \rightarrow \sigma$ and b) $\Gamma \vdash e_2 : \sigma$. We also have that c) $e_1; S \downarrow v; S'$ which by induction hypothesis lets us conclude that there is a Γ' typing S' such that $\Gamma \vdash v : \tau \rightarrow \sigma$ and v is an abstraction. Therefore if an application expression is well-typed, the Rule (Wrong Call) is never applicable.

Case: **(Wrong Assign)**

If $\Gamma \vdash e_1.\ell := e_2 : \tau$ then by Rule (Eval Assign) we know that a) $\Gamma \vdash e_1 : \{\!| \ldots, \ell : \tau, \ldots |\!\}$. The evaluation hypothesis has the premise b) $e_1; S \downarrow v; S'$. Taking a) and b), by induction hypothesis we know that there is Γ' typing S' and extending Γ such that $\Gamma \vdash v : \{\!|\ \ldots, \ell : \tau, \ldots\ |\!\}$ and v must be a location leading to a record. Thus Rule (Wrong Assign) is never applicable to well-typed assignment expressions.

***Case:* (Wrong Assign 2)**

Following the reasoning from the previous case, we have $\Gamma \vdash \imath : \{\!|\ \ldots, \ell : \tau, \ldots\ |\!\}$ and that $\imath$ is a location leading to a record. By Definition 2.21 we know that $\Gamma \vdash S(\imath) : \{\!|\ \ldots, \ell : \tau, \ldots\ |\!\}$ which by Rule (Val Record) must contain the label ℓ and therefore Rule (Wrong Assign 2) is also never applicable.

***Case:* (Wrong Select)**. Similar.

***Case:* (Wrong Select 2)**. Similar.

***Case:* (Wrong Compose)**

If $\Gamma \vdash \mathsf{compose}\ e : K_{\lhd} \Rightarrow K_{\rhd}$ then we know that $\Gamma \vdash e : \emptyset \Longrightarrow K$ by induction hypothesis we have a Γ' extends Γ and types S' and that v is a value such that $\Gamma \vdash v : \emptyset \Longrightarrow K$. This value can only be a configurator value. So wrong is never issued by Rule (Wrong Compose).

***Case:* (Wrong New)**

If $\Gamma \vdash \mathsf{new}\ e\ \mathsf{with}\ \ell^r_j := e_j\ ^{j \in 1..m} : \sigma$, by Rule (Val New), we have that $\Gamma \vdash e : \tau \Rightarrow \sigma$ and that $e; S \downarrow v; S'$. By induction hypothesis we obtain that there is a Γ' extending Γ and typing S' such that v is a component value (other possible values have different types). This is contradicts the application of (Wrong New).

***Case:* (Wrong Sequence)**

If $\Gamma \vdash e_1; e_2 : K, K'' \Longrightarrow K', K'''$ then $\Gamma \vdash e_1 : K \Longrightarrow K', K_c$ and, as $e_1; S \downarrow v; S$ we know, by induction hypothesis, that there is Γ' extending Γ and typing S' such that v is a configurator value with $\Gamma' \vdash v : K \Longrightarrow K', K_c$. Once again, we find a contradiction.

***Case:* (Wrong Sequence 2)**. Similar.

The second part of the lemma states that the application of a configurator to an instance causes an effect consistent with the type of the operation. This proof is done by induction on the height of the derivations and by case analysis of the last evaluation rule used. We use the first part of the lemma when necessary.

***Case:* (App Provides)**

In this case we have an expression typed $\Gamma \vdash (\mathsf{provides}\ \ell : \tau) : \emptyset \Longrightarrow \{\ell \circ \tau, \ell \rhd \tau\}$ and an instance $s = (r, e, p)$ that conforms with $\emptyset$. The resulting instance, $(r, e, p \oplus \{\ell = \imath\})$, trivially conforms, by Definition 3.23, with the resource set $\{\ell \circ \tau, \ell \triangleright \tau\}$ and $\Gamma' = \Gamma, \imath : \tau$ types the heap $S[\imath \mapsto \mathsf{nil}]$.

Since the set of demanded resources in the configurator type is empty we know that $\mathsf{nil}(S) \subseteq X$ for some X. By Rule (App Provides) we know that $\text{select}_{S'}(s', \ell) = \imath$ and $S'(\imath) = \mathsf{nil}$. More, $\imath$ is the only new location in S' with relation to S and $K'_{\triangleleft} \cup K'_{\circ} = \{\ell : \tau\}$. So, $\{\text{select}_{S'}(s', \ell) \mid (\ell : \tau) \in K'_{\triangleleft} \cup K'_{\circ}\} = \{\imath\}$ and therefore $\mathsf{nil}(S') \subseteq \{\imath\} \cup X$ which corresponds to the expected results. With relation to the final instance type, we have that if the type of s is $[\![\tau \Rightarrow \sigma]\!]$ then the partial type of s' is $[\![\tau \Rightarrow \sigma \oplus \{\ell : \tau\}]\!]$. Since no record is introduced in the heap $S[\imath \mapsto \mathsf{nil}]$ is still record-based.

***Case:* (App Requires)**

In this case we have the typing judgment $\Gamma \vdash (\mathsf{requires}\ \ell : \tau) : \emptyset \Longrightarrow \{\ell \bullet \tau, \ell \triangleleft \tau\}$ as hypothesis and an instance $s = (r, e, p)$ that conforms with $\emptyset$. The resulting instance, $s' = (r \oplus \{\ell = \imath\}, e, p)$, trivially conforms, by Definition 3.23, with the resource set $\{\ell \bullet \tau, \ell \triangleleft \tau\}$ and the typing environment $\Gamma' = \Gamma, \imath : \tau$, which also types the heap $S' = S[\imath \mapsto \mathsf{nil}]$. The resulting value is well-typed according to Rule (Val Object).

Since the set of demanded resources in the configurator type is empty we have that $\mathsf{nil}(S) \subseteq X$ for some X. By Rule (App Requires) we know that $\text{select}_{S'}(s', \ell) = \imath$ and that $S'(\imath) = \mathsf{nil}$. More, $\imath$ is the only new location in S' with relation to S and $K'_{\triangleleft} \cup K'_{\circ} = \{\ell : \tau\}$. So, $\{\text{select}_{S'}(s', \ell) \mid (\ell : \tau) \in K'_{\triangleleft} \cup K'_{\circ}\} = \{\imath\}$ and therefore $\mathsf{nil}(S') \subseteq \{\imath\} \cup X$ which corresponds to the expected results. With relation to the partially linked object type, we have that if the initial type of s is $[\![\tau \Rightarrow \sigma]\!]$ then the partial type of s' is $[\![\tau \oplus \{\ell : \tau\} \Rightarrow \sigma]\!]$ and the resulting heap is record-based.

***Case:* (App Uses)**

The expression $x[v : \tau \Rightarrow \sigma]$ where $\tau = \{\ell^r_i : \tau_i\ ^{i \in 1..n}\}$ and $\sigma = \{\ell^p_j : \sigma_j\ ^{j \in 1..m}\}$ is typed by a judgement $\Gamma \vdash x[v : \tau \Rightarrow \sigma] : \emptyset \Longrightarrow K$ with $K = \{x \bullet \sigma, x.\ell^r_i \circ \tau'_i\ ^{i \in 1..n}, x.\ell^p_j \bullet \sigma'_j\ ^{j \in 1..m}\}$. By Rule (Comp Uses) we have that $\Gamma \vdash v : \tau \Rightarrow \sigma$. Rule (Val Composition Value) is the only one that types a component value and therefore we have $v = \mathsf{comp}(c)$ with $\Gamma \vdash c : \emptyset \Longrightarrow K'$ with $K'_{\circ} = \emptyset$ and $(\tau \Rightarrow \sigma) = (K'_{\triangleleft} \Rightarrow K'_{\triangleright})$.

From the application judgement, $s; x[v : \tau \Rightarrow \sigma]; S \Downarrow s'; S'$, we know that $(\mathsf{new}\ v); S \downarrow \imath; S'$. Since v only evaluates to itself with no changes to the heap, and there are no plug-assignments, the evaluation of this **new** expression depends solely on $\mathbf{0}; c; S \Downarrow s; S''$ with $S' = S''[\imath \mapsto s]$.

Since $\Gamma \vdash x[v : \tau \Rightarrow \sigma] : \emptyset \Longrightarrow K$, where the set of demanded resources is empty, we know that there is a set X such that $\mathsf{nil}(S) \subseteq X$. By induction hypothesis on the height of the evaluation derivation we have that there is a Γ' that types S'' and that the resulting value $s = (r', e', p')$ conforms with K' with the partially linked object type $[\![K'_{\triangleleft} \Longrightarrow K'_{\triangleright}]\!]$ with relation to Γ''. We know that $s' = (r, e \oplus \{x \mapsto \imath\}, p)$ and the store S' are typed by $\Gamma'' = \Gamma', \imath :$

${K'}_{\triangleright}$. Notice that the instance type corresponding to $[\![\tau \Rightarrow \sigma]\!]$ is ${K'}_{\triangleright}$. More, the instance s' conforms, according to Definition 3.23, with the resource set K. By the induction hypothesis we also know that $\mathsf{nil}(S'') \subseteq \{\mathrm{select}_{S''}(s,\ell) \mid (\ell : \tau) \in {K'}_{\triangleleft}\} \cup X$ with ${K'}_{\triangleleft} = \{\ell^r_i : \tau_i{}^{i \in 1..n}\}$. We then conclude that, in the context of the containing instance, $\mathsf{nil}(S') \subseteq \{\mathrm{select}_{S''}(s',\pi) \mid (\pi : \tau) \in \{x.\ell^r_i : \tau_i{}^{i \in 1..n}\}\} \cup X$. The partial type of the resulting object remains unchanged since no required or provided ports are added to the instance.

Case: **(App Method Block)**

If $\Gamma \vdash x_K[\ell_i : \tau_i = v_i{}^{i \in 1..n}] : K \implies K, \{x \bullet \{\ell_i : \tau_i{}^{i \in 1..n}\}\}$, then by Rule (Comp Method Block), we know that for all $i \in 1..n$, $|\Gamma|, K_\bullet, x : \{\!|\ell_i : \tau_i{}^{i \in 1..n}|\!\} \vdash v_i : \tau_i$. Since (r,e,p) conforms with K and $v_i[(r,e,p)][x \leftarrow \imath]$ denotes the substitution of the available names in the current instance (r,e,p) by its locations, which are typed in Γ, by Lemma 3.24 (weakening) and Lemma 3.28 (substitution), we obtain $\Gamma, x : \{\!|\ell_i : \tau_i{}^{i \in 1..n}|\!\} \vdash v_i[(r,e,p)] : \tau_i$ for all $i \in 1..n$. Let $\Gamma' = \Gamma, \imath : \{\!|\ell_i : \tau_i{}^{i \in 1..n}|\!\}$ such that by Lemma 3.24 (weakening) and 3.28 (substitution) we have that $\Gamma' \vdash v_i[(r,e,p)][x \leftarrow \imath] : \tau_i$ for all $i \in 1..n$. So, we have that $\Gamma', \imath_i : \tau_i{}^{i \in 1..n}$ types the resulting heap $S' = S[\imath \mapsto \{\ell_i \mapsto \imath_i{}^{i \in 1..n}\}][\imath_i \mapsto v_i[(r,e,p)][x \leftarrow \imath]{}^{i \in 1..n}]$.

Since (r,e,p) conforms with K this makes the instance, $(r, e \oplus \{x = \imath\}, p)$, conform with $K' = K, \{x \bullet \{\ell_i : \tau_i{}^{i \in 1..n}\}\}$. No references are created in the heap that lead to nil, hence $\mathsf{nil}(S') = \mathsf{nil}(S)$, and by considering the set X such that $\mathsf{nil}(S) \subseteq \{\mathrm{select}_S(s,\ell) \mid (\ell : \tau) \in K_\triangleleft \cup K_\circ\} \cup X$ we have that $\mathsf{nil}(S') \subseteq \{\mathrm{select}_S(s,\ell) \mid (\ell : \tau) \in {K'}_\triangleleft \cup {K'}_\circ\} \cup X$ which is the same set. As in the previous case, the partial object type remains unchanged. Notice that the original expressions v_i are record based and when replacing the locations of the object (which are also record-based) the resulting values which are placed in the new record in the heap are record-based.

Case: **(App Plug)**

If $\Delta \vdash \mathsf{plug}\ (\pi_1 : \tau)\ \mathsf{into}\ (\pi_2 : \tau) : (\{\pi_2 \circ \tau, \pi_1 \bullet \tau\} \implies \{\pi_1 \bullet \tau\})$ only the unsatisfied inner requirements change, So ${K'}_\circ \subseteq K_\circ$, with $K = \{\pi_2 \circ \tau, \pi_1 \bullet \tau\}$ and $K' = \{\pi_1 \bullet \tau\}$. However, the resulting instance is the same and by Definition 3.23 s conforms with K'. On the other hand, the heap is changed to make the connection between the source and the target of the plug operation. From the lemma's hypothesis we know that $\mathsf{nil}(S) \subseteq \{\mathrm{select}_S(s,\ell) \mid (\ell : \tau) \in K_\triangleleft \cup K_\circ\} \cup X$. So, $S(\mathrm{select}_S(s,\pi_2)) = \mathsf{nil}$ and therefore $\mathsf{nil}(S[\mathrm{select}_S(s,\pi_2) \mapsto \mathrm{select}_S(s,\pi_1)]) \subseteq \{\mathrm{select}_S(s,\ell) \mid (\ell : \tau) \in {K'}_\triangleleft \cup {K'}_\circ\} \cup X$. This operation links two ports whose types are interface types, so the heap stays record-based.

Case: **(App Sequence)**

We first prove that the resulting instance conforms with the type. The typing judgement $\Delta \vdash e_1; e_2 : K, K'' \implies K', K'''$ is supported by the premises $\Delta \vdash e_1 : K \implies K', K_c$ and $\Delta \vdash e_2 : K'_c, K'' \implies K'''$. By induction hypothesis on the second case of the lemma, together with the application of e_1 to an instance s which conforms with K we reach an instance s' which conforms with K', K_c. Since s' extends s it also conforms with K'' and by Lemma 3.24 (weakening) and induction hypothesis on the second case of the lemma together with the evaluation of e_2 we conclude that s'' conforms with K'''. Since it extends s and s' we conclude that it also conforms with K'.

Now concerning the nil values in the heap. In the case of $s; (c_1; c_2); S \Downarrow s''; S''$ with $\Delta \vdash (e_1; e_2) : K, K'' \Longrightarrow K', K'''$ we have that there is a set X such that $\mathsf{nil}(S) \subseteq \{\mathrm{select}_S(s, \ell) \mid (\ell : \tau) \in (K, K'')_\triangleleft \cup (K, K'')_\circ\} \cup X$. By rewriting this definition we obtain $\mathsf{nil}(S) \subseteq \{\mathrm{select}_S(s, \ell) \mid (\ell : \tau) \in K_\triangleleft \cup K_\circ\} \cup X'$ with $X' = \{\mathrm{select}_S(s, \ell) \mid (\ell : \tau) \in K''_\triangleleft \cup K''_\circ\} \cup X$. Given the application of the first premise $s; c_1; S \Downarrow s'; S'$ typed $\Delta \vdash e_1 : K \Longrightarrow K', K_c$, by induction hypothesis, we know that $\mathsf{nil}(S') \subseteq \{\mathrm{select}_S(s, \ell) \mid (\ell : \tau) \in (K', K_c)_\triangleleft \cup (K', K_c)_\circ\} \cup X'$. Which is the same as $\mathsf{nil}(S') \subseteq \{\mathrm{select}_S(s, \ell) \mid (\ell : \tau) \in K'_\triangleleft \cup K'_\circ\} \cup \{\mathrm{select}_S(s, \ell) \mid (\ell : \tau) \in K_{c\triangleleft} \cup K_{c\circ}\} \cup \{\mathrm{select}_S(s, \ell) \mid (\ell : \tau) \in K''_\triangleleft \cup K''_\circ\} \cup X$. Again, by rearranging the definition we have $\mathsf{nil}(S') \subseteq \{\mathrm{select}_S(s, \ell) \mid (\ell : \tau) \in K'', K_{c\triangleleft} \cup K'', K_{c\circ}\} \cup X''$ with $X'' = \{\mathrm{select}_S(s, \ell) \mid (\ell : \tau) \in K'_\triangleleft \cup K'_\circ\} \cup X$. Given the second premise evaluated by $s; c_1; S \Downarrow s'; S'$ and typed by $\Delta \vdash e_2 : K_c, K'' \Longrightarrow K'''$, by induction hypothesis we know that $\mathsf{nil}(S'') \subseteq \{\mathrm{select}_S(s, \ell) \mid (\ell : \tau) \in K', K'''_\triangleleft \cup K', K'''_\circ\} \cup X''$. As the locations in X'' corresponding to the labels in K' are already in this final set we conclude that $\mathsf{nil}(S'') \subseteq \{\mathrm{select}_S(s, \ell) \mid (\ell : \tau) \in K', K'''_\triangleleft \cup K', K'''_\circ\} \cup X$.

Case: **(Wrong Plug)**

In this case, we would have that $\mathrm{select}_S(s, \pi_1)$ is undefined. By Definition 3.7 we know that either $\pi_1 = x$ or $\pi_1 = x.\ell$.

Subcase: $\pi_1 = x$

In this case we know that s conforms with $\{x \bullet \tau, \pi_2 \circ \tau\}$ and by Definition 3.23 we know that $s = (r, e, p)$ and that $r \oplus e = \{\ldots, x = \imath, ...\}$. Hence, $\mathrm{select}_S(s, \pi_1)$ must be defined.

Subcase: $\pi_1 = x.\ell$

In this case we know that s conforms with $\{x.\ell \bullet \tau, \pi_2 \circ \tau\}$ and by Definition 3.23 we know that $s = (r, e, p)$ and that $e = \{\ldots, x = \imath, \ldots\}$ and that $S(\imath) = (r', e', p')$ and that $p' = \{\ldots, \ell = \imath', \ldots\}$. Hence, $\mathrm{select}_S(s, \pi_1)$ must be defined.

In both cases the rule is not applicable.

Case: **(Wrong Uses)**

If the expression is well-typed, we know that v is a value of type component (by Rule (Comp Uses)) and therefore the rule is not applicable. □

A.2 Chapter 3

In this section we give the detailed proofs of chapter 4.
(This is a repetition of Lemma 4.9, defined in page 97.)

Lemma 4.9. *For all instances s and resource sets K, if $s\,/\!/\,K$ then s conforms with K.*

Proof. The proof is done by induction on the height of the derivations of $s\,/\!/\,K$ with $s = (r, e, p)_\Gamma$ and by case analysis on the last rule used.

***Case:* (Match Provides)**

In this case we have that $s\,/\!/\,(\ell \triangleright \tau, K)$ and, by Rule (Match Provides), we have that $\Gamma(s_\triangleright.\ell) = \tau$ and $s/\!/K$ with $s_\triangleright = p$. By induction hypothesis we know that s conforms with K. Therefore conclude, by Definition 3.23 (compliance), that $K_\triangleright \subseteq p$ with $p = \{\ell^p_i : \tau^p_i \,{}^{i\in 1..n^p}\}$. From $\Gamma(s_\triangleright.\ell) = \tau$ we know that ℓ must be one of the provided ports ℓ^p_i with $i \in 1..n^p$. So, we conclude that $(\ell : \tau, K_\triangleright) \subseteq \{\ell^p_i : \tau^p_i \,{}^{i\in 1..n^p}\}$ and that s conforms with $(\ell \triangleright \tau, K)$.

***Case:* (Match Requires)**

The result $s\,/\!/\,(\ell \triangleleft \tau, K)$ is supported, by Rule (Match Requires), that $\Gamma(s_\triangleleft.\ell) = \tau$ and that $s/\!/K$ with $s_\triangleleft = r$. By induction hypothesis we know that s conforms with K. Therefore we conclude, by Definition 3.23 (conformance), that $K_\triangleleft \subseteq r$ with $r = \{\ell^r_i : \tau^r_i \,{}^{i\in 1..n^r}\}$. From $\Gamma(s_\triangleleft.\ell) = \tau$ we conclude that ℓ must be one of the required ports ℓ^r_i with $i \in 1..n^r$. So, we conclude that $(\ell : \tau, K_\triangleleft) \subseteq \{\ell^r_i : \tau^r_i \,{}^{i\in 1..n^r}\}$ and that s conforms with $(\ell \triangleleft \tau, K)$.

***Case:* (Match Element)**

If the last rule used is (Match Element), then $s/\!/(\ell \bullet \tau, K)$ is supported by $\Gamma(s_\bullet.\ell) = \tau$ and that $s/\!/K$ with $s_\triangleleft = r$. By induction hypothesis we know that s conforms with K. In this case, we conclude, by Definition 3.23 (conformance) that the available resources are one of the required ports, or one of the method blocks, one of the inner elements or a provided port of an inner element: $K_\bullet \subseteq \{\ell^r_i : \tau^r_i \,{}^{i\in 1..n^r}\} \cup \{\ell^m_i : \tau^m_i \,{}^{i\in 1..n^m}\} \cup \{\ell^c_j.\ell^{pj}_h : \gamma^j_h \,{}^{j\in 1..n^c}\,{}^{h\in 1..m^c_j}, \ell^c_i : \{\ell^{pi}_h : \gamma^i_h \,{}^{h\in 1..m^c_i}\}^{i\in 1..n^c}\}$. From $\Gamma(s_\bullet.\ell) = \tau$ we conclude that ℓ must be one of the required ports ℓ^r_i with $i \in 1..n^r$, one of the method blocks ℓ^m_i with $i \in 1..n^m$, or one of the inner elements ℓ^c_i with $i \in 1..n^c$. Hence, s conforms with $(\ell \bullet \tau, K)$.

***Case:* (Match Unsatisfied)**

If the last rule is (Match Unsatisfied) then $s\,/\!/\,(\ell \circ \tau, K)$ is supported by $\Gamma(s_\circ.\ell) = \tau$ and that $s/\!/K$ with $s_\circ = r$. By induction hypothesis we know that s conforms with K and, by Definition 3.23 (conformance), we know that $K_\circ \subseteq \{\ell^p_i : \tau^p_i \,{}^{i\in 1..n^p}\} \cup \{\ell^c_j.\ell^{rj}_h : \delta^j_h \,{}^{j\in 1..n^c}\,{}^{h\in 1..n^c_j}\}$. From $\Gamma(s_\circ.\ell) = \tau$ we conclude that ℓ must be one of the provided ports ℓ^p_i with $i \in 1..n^p$. Hence, s conforms with $(\ell \circ \tau, K)$.

Case: **(Match Element Port)**

In the case of $s \,/\!/\, (x.\ell \bullet \tau, K)$, by Rule (Match Element Port) we know that $s \,/\!/\, K$ and therefore, by induction hypothesis, we know that it conforms with K. Thus, by Definition 3.23, we have that $K_\bullet \subseteq \{\ell^r_i : \tau^r_i \;{}^{i\in 1..n^r}\} \cup \{\ell^m_i : \tau^m_i \;{}^{i\in 1..n^m}\} \cup \; \{\ell^c_j.\ell^{pj}_h : \gamma^j_h \;{}^{j\in 1..n^c \;\; h\in 1..m^c_j}, \ell^c_i : \{\ell^{pi}_h : \gamma^i_h \;{}^{h\in 1..m^c_i}\}^{i\in 1..n^c}\}$.

We also know, from Rule (Match Element Port), that $s_\bullet = \{\ldots, x = \imath, \ldots\}$, $S(\imath) = s'$, and $\Gamma'(s'_\triangleright.\ell) = \tau$. Then we have that x is one of the inner components ℓ^c_j with $i \in 1..n^c$, and that $x.\ell$ is one of its provided ports the $\ell^c_j.\ell^{pj}_h$ with $\tau = \gamma^j_h$ and $j \in 1..n^c$ and $h \in 1..m^c_j$. This means that $K_\bullet \subseteq \{\ell^r_i : \tau^r_i \;{}^{i\in 1..n^r}\} \cup \{\ell^m_i : \tau^m_i \;{}^{i\in 1..n^m}\} \cup \; \{\ell^c_j.\ell^{pj}_h : \gamma^j_h \;{}^{j\in 1..n^c \;\; h\in 1..m^c_j}, \ell^c_i : \{\ell^{pi}_h : \gamma^i_h \;{}^{h\in 1..m^c_i}\}^{i\in 1..n^c}\}$ and that s conforms with $(x.\ell \bullet \tau, K_\bullet)$.

Case: **(Match Unsatisfied Port)**

In the case of $s \,/\!/\, (x.\ell \circ \tau, K)$, by Rule (Match Unsatisfied Port) we know that $s \,/\!/\, K$ and therefore, by induction hypothesis, we know that it conforms with K. Thus, by Definition 3.23, we have that $K_\circ \subseteq \{\ell^p_i : \tau^p_i \;{}^{i\in 1..n^p}\} \cup \{\ell^c_j.\ell^{rj}_h : \delta^j_h \;{}^{j\in 1..n^c \;\; h\in 1..n^c_j}\}$.

We also know, from Rule (Match Unsatisfied Port), that $s_\bullet = \{\ldots, x = \imath, \ldots\}$, $S(\imath) = s'$, and $\Gamma'(s'_\triangleleft.\ell) = \tau$. Then we have that x is one of the inner components ℓ^c_j with $i \in 1..n^c$, and that $x.\ell$ is one of its required ports the $\ell^c_j.\ell^{rj}_h$ with $\tau = \delta^j_h$ and $j \in 1..n^c$ and $h \in 1..n^c_j$. This means that $(x.\ell \circ \tau, K_\circ) \subseteq \{\ell^p_i : \tau^p_i \;{}^{i\in 1..n^p}\} \cup \{\ell^c_j.\ell^{rj}_h : \delta^j_h \;{}^{j\in 1..n^c \;\; h\in 1..n^c_j}\}$ and that s conforms with $(x.\ell \circ \tau, K_\circ)$. □

(This is a repetition of Lemma 4.12, defined in page 98.)

Lemma 4.12 (Subject Reduction).

1. *Let $(e; S)$ be a valid configuration in $\lambda_\rho \backslash \{\mathsf{nil}\}$ such that $\mathsf{nil}(S) = \emptyset$ and let Γ be a typing environment typing S such that e and S are record-based with relation to Γ.:*

   If $\Gamma \vdash e : \tau$ and $(e; S \downarrow v; S')$ then

   a) *there is a Γ' that extends Γ and types S',*

   b) *$\Gamma' \vdash v : \tau$,*

   c) *v is either an abstraction, a component, a configurator, or a location that is either `undefined` or `refers-to` a record,*

   d) *v and S' are record-based with relation to Γ, and*

   e) *$\mathsf{nil}(S') = \emptyset$.*

2. *Let c be an expression such that $\Gamma \vdash c : K \Longrightarrow K'$, let S be a heap such that, for some set $X \in Dom(S)$, $\mathsf{nil}(S) \subseteq \{\mathsf{select}_S(s, \pi) \mid (\pi : \tau) \in K_\triangleleft \cup K_\circ\} \uplus X$ and Γ types S.*

Let s be a partially linked object s such that it conforms with K and its partially linked object type is $[\![R \oplus K_{\triangleleft} \Rightarrow P \oplus K_{\triangleright}]\!]$ and s and S are record-based with relation to Γ:

If $s; c; S \Downarrow s'; S'$ then

a) *there is Γ' typing S' and extending Γ,*

b) *s' is a partially linked object that extends s and conforms with K'. Its partially linked object type is $[\![R \oplus K'_{\triangleleft} \Rightarrow P \oplus K'_{\triangleright}]\!]$, and*

c) *s' and S' are record-based with relation to Γ', and*

d) $\mathsf{nil}(S') \subseteq \{\mathrm{select}_{S'}(s, \pi) \mid (\pi : \tau) \in K'_{\triangleleft} \cup K'_{\circ}\} \uplus X$.

Proof. The proof is carried out by induction on the two cases of the lemma. On the first case, we prove it by induction on the size of the evaluation derivations and by analysis of the last rule used. We use the second case when needed and its conditions are met. We show, in the cases that evaluate to wrong, that these rules are never applicable.

***Case:* (Eval Value)**

The conclusions of the theorem hold with $\Gamma' = \Gamma$, $S' = S$, and $v = \lambda x : \tau.e$. We also have that $\mathsf{nil}(S') = \emptyset$.

***Case:* (Eval Application)$^{\tau}$**

In this case, with $e_1(e_2); S \downarrow v; S'$, we know by Rule (Eval Application), that: a) $e_1; S \downarrow \lambda x : \tau.e; S'$, b) $e_2; S' \downarrow v_2; S''$, and c) $e[x \leftarrow v_2]; S'' \downarrow v; S'''$. On the typing derivation, the only plausible last rule for $\Gamma \vdash e_1(e_2) : \sigma$ is Rule (Val Application), with the premises: d) $\Gamma \vdash e_1 : \tau \rightarrow \sigma$ and e) $\Gamma \vdash e_2 : \tau$.

Taking d) and a), by induction hypothesis we conclude that there is a Γ' extending Γ and typing S', such that f) $\Gamma' \vdash \lambda x : \tau.e : \tau \rightarrow \sigma$. By Lemma 3.24 (weakening) on e) we conclude that $\Gamma' \vdash e_2 : \tau$, which together with b), by induction hypothesis, there is a Γ'' extending both Γ' and Γ and typing S'' such that g) $\Gamma'' \vdash v_2 : \tau$ and $\mathsf{nil}(S'') = \emptyset$. From f), by Lemma 3.24 (weakening) and Rule (Val Abstraction) we have that $\Gamma'', x : \tau \vdash e : \sigma$ and by Lemma 3.28 (substitution) with g) we have that h) $\Gamma'' \vdash e[x \leftarrow v_2] : \sigma$. Taking h) and c), by induction hypothesis we have that there is Γ''' extending Γ'' and typing S''', such that v is typed $\Gamma''' \vdash v : \sigma$ and v is either an abstraction or a location that is either *undefined* or *refers-to* a record, and $\mathsf{nil}(S''') = \emptyset$.

The application of induction hypothesis to the different premises and the application of Lemma 3.28 ensures that v and S''' are record-based.

***Case:* (Eval Record)**

When the last rule is (Eval Record) we have the hypothesis $[\ell_i = e_i\ ^{i \in 1..n}]; S_0 \downarrow \imath; S'$ with $S' = S_n[\imath_i \mapsto v_i\ ^{i \in 1..n}][\imath \mapsto \{\ell_i = \imath_i\ ^{i \in 1..n}\}]$ and $\Gamma \vdash [\ell_i = e_i\ ^{i \in 1..n}] : \{\!|\ell_i : \tau_i\ ^{i \in 1..n}|\!\}$. By the Rules (Val Record) and (Eval Record), we have that a) $\Gamma \vdash e_i : \tau_i$ and b) $e_i; S_{i-1} \downarrow v_i; S_i$ with $i \in 1..n$. By iterating the induction hypothesis with a) and b), and using Lemma 3.24 (weakening) to adjust the typing environment in the hypothesis, we reach the conclusion that, for all $i \in 1..n$, there is a Γ_i extending Γ_{i-1} and typing each S_i, with $\Gamma_0 = \Gamma$ such that $\Gamma_i \vdash v_i : \tau_i$ and $\mathsf{nil}(S_i) = \emptyset$. So, by transitivity, we have that Γ_n types S_n and extends Γ with $\mathsf{nil}(S_n) = \emptyset$. By Rule (Val Record Value) and Definition 2.21 we have that $\Gamma' = \Gamma_n, \imath : \{\!|\ell_i : \tau_i\ ^{i \in 1..n}|\!\}$ types S' and that $\Gamma' \vdash \imath : \{\!|\ell_i : \tau_i\ ^{i \in 1..n}|\!\}$ with relation to S'. We have that $\imath$ is a location that *refers-to* a record and $\mathsf{nil}(S') = \emptyset$. The resulting heap is also record-based since all the introduced values are record-based (by induction hypothesis).

***Case:* (Eval Assign)**

In this case, from $e_1.\ell := e_2$, by Rule (Eval Assign), we have that a) $e_1; S \downarrow \imath; S'$ with $\imath' = \mathrm{deref}_{S'}(\imath)$ and $S'(\imath') = \{\dots, \ell = \imath'', \dots\}$, and b) $e_2; S' \downarrow v; S''$. By Rule (Val Assign), we have that $\Gamma \vdash (e_1.\ell := e_2) : \sigma$ is supported by c) $\Gamma \vdash e_1 : \{\!|\dots, \ell : \sigma, \dots|\!\}$ and d) $\Gamma \vdash e_2 : \sigma$. Thus, from a) and c), by induction hypothesis we know that there is a Γ' extending Γ and typing S' such that $\Gamma' \vdash \imath : \{\!|\dots, \ell : \sigma, \dots|\!\}$. and $\mathsf{nil}(S') = \emptyset$. By Definition 2.21 we conclude that $S'(\mathrm{deref}_{S'}(\imath)) = \{\dots, \ell = \imath'', \dots\}$ and that $\Gamma \vdash S'(\mathrm{deref}_{S'}(\imath)) : \{\!|\dots, \ell : \sigma, \dots|\!\}$. By rule (Val Record) we conclude that e) $\Gamma' \vdash \imath'' : \sigma$. From b), by Lemma 3.24 (weakening) we obtain $\Gamma' \vdash e_2 : \sigma$, and by induction hypothesis together with d) we have that there is a Γ'' extending Γ' and typing S'' such that f) $\Gamma'' \vdash v : \sigma$. and $\mathsf{nil}(S'') = \emptyset$. Γ' types $S''[\imath \mapsto v]$ and the result of the assignment expression is v, and v is not nil we conclude this case with $\mathsf{nil}(S''[\imath \mapsto v]) = \emptyset$. with v satisfying the conclusions of the theorem. Value v is record-based and it is introduced inside a record, so the resulting heap $S''[\imath \mapsto v]$ is also record-based.

***Case:* (Eval Select)**

If the last rule used is (Eval Select) then the expression $e.\ell$ can be typed by two possible rules. In both cases we have that a) $e_1; S \downarrow \imath; S'$ with $\imath' = \mathrm{deref}_{S'}(\imath)$ and $S'(\imath') = \{\dots, \ell = \imath'', \dots\}$. The subcases are:

***Subcase:* (Val Select)**

Now, in this case we have that c) $\Gamma \vdash e : \{\!|\dots, \ell : \tau, \dots|\!\}$. From a) and c), by induction hypothesis we know that there is a Γ' extending Γ and typing S' such that $\Gamma' \vdash \imath : \{\!|\dots, \ell : \sigma, \dots|\!\}$ and $\mathsf{nil}(S') = \emptyset$. By Definition 2.21 we conclude that for some $\imath' = \mathrm{deref}_{S'}(\imath)$ we have $S'(\imath') = \{\dots, \ell = \imath'', \dots\}$ and that $\Gamma \vdash S'(\imath') : \{\!|\dots, \ell : \sigma, \dots|\!\}$. By Rule (Val Record) we conclude that e) $\Gamma' \vdash \imath'' : \sigma$. By Definition 2.21 we conclude that $\Gamma' \vdash S'(\imath'') : \sigma$. Notice that $\mathsf{nil}(S') = \emptyset$. The resulting value results from consulting a record in a record-based heap, so, it is also record-based.

***Subcase:* (Val Select Interface)**

In this case we know that c) $\Gamma \vdash e : \{\dots, \ell : \tau, \dots\}$ and therefore, by induction hypothesis (from

a) and c)), that e_1 yields an interface typed value (l). By inspection of the type system (Definition 3.18) we see, by Rule (Val Select Interface), that $\Gamma \vdash \mathit{l} : \{\!| \ldots, \ell : \tau, \ldots |\!\}$ and then the reasoning follows by applying Definition 2.21 and Rule (Val Record) as in the subcase above.

Case: **(Eval Compose)**

We have the typing $\Gamma \vdash \mathsf{compose}\ e : K_\triangleleft \Rightarrow K_\triangleright$ supported by $\Gamma \vdash e : \emptyset \Longrightarrow K$ where $K_\circ = \emptyset$. The evaluation hypothesis is $\mathsf{compose}\ e; S \downarrow \mathsf{comp}(c); S'$ where $e; S \downarrow \mathsf{conf}(\tau, c); S'$. By induction hypothesis we have that there is a Γ' extending Γ, that types S' and $\Gamma' \vdash \mathsf{conf}(\tau, c) : \emptyset \Longrightarrow K$ and $\mathsf{nil}(S') = \emptyset$. By Rule (Val Configurator Value)$^\rho$ we know that $\Gamma' \vdash c : \tau$ and that $\tau = \emptyset \Longrightarrow K$, and by Rules (Val Compose) and (Val Composition Value) we conclude that $\Gamma' \vdash \mathsf{comp}(c) : K_\triangleleft \Rightarrow K_\triangleright$. with Γ' extending Γ and typing S', and with the resulting value being a component value. From the application of the induction hypothesis we also conclude that $\mathsf{nil}(S') = \emptyset$ and that the value $comp(c)$ is record-based.

Case: **(Eval New)**

The hypothesis $\Gamma \vdash \mathsf{new}\ e\ \mathsf{with}\ \ell^r_j := e_j\ ^{j \in 1..n} : \{\ell^p_i : \sigma_i\ ^{i \in 1..n}\}$ is obtained by Rule (Val New) with the premises $\Gamma \vdash e : \tau$ with $\tau = \{\ell^r_j : \tau_j\ ^{j \in 1..m}\} \Rightarrow \{\ell^p_i : \sigma_i\ ^{i \in 1..n}\}$. new is evaluated by Rule (Eval New), thus, if $\mathsf{new}\ e\ \mathsf{with}\ \ell^r_j := e_j\ ^{j \in 1..m}; S \downarrow \mathit{l}; S'$, then we must have $e; S \downarrow \mathsf{comp}(c); S_0$. By induction hypothesis we have that there is a Γ_0 extending Γ such that $\Gamma_0 \vdash \mathsf{comp}(c) : \tau$ and $\mathsf{nil}(S_0) = \emptyset$.

Now, notice that the presentation order of the premises does not reflect the real dependence between them. For technical reasons, the plug-assignments must be considered first in this proof. By iteratively applying Lemma 4.11 (substitution), the induction hypothesis, and Lemma 4.10 (weakening) on the typing and evaluation judgements of each plug-assignment expression, ($\Gamma \vdash e_i : \tau_i$) and ($e_i; S_{i-1} \downarrow v_i; S_i$), we obtain that for all $i \in 1..n$ we have Γ_i extending Γ_{i-1} and typing S_i such that $\Gamma_i \vdash v_i : \tau_i$ and $\mathsf{nil}(S_i) = \emptyset$. Notice that all v_i and S_i are record-based.

By Lemma 4.10 (weakening) on the typing of $\mathsf{comp}(c)$ and Rule (Val Composition Value) we have $\Gamma_n \vdash c : \emptyset \Longrightarrow K$ and $K_\circ = \emptyset$, with type $\tau = K_\triangleleft \Rightarrow K_\triangleright$ thus $K_\triangleleft = \{\ell_j : \tau_j\ ^{j \in 1..m}\}$. Notice that the judgement above applies c to the empty instance ($\mathbf{0}; c; S_n \Downarrow s; S_{n+1}$), and that $\mathsf{nil}(S_n) = \emptyset$ meets the conditions of the second part of the lemma with $X = \emptyset$. In this case, $\mathbf{0}$ conforms with the resource list $\emptyset$ and has type $[\![\{\} \Rightarrow \{\}]\!]$. By induction hypothesis on the second case of the lemma, we have that there is a Γ_{n+1} extending Γ_n and the resulting instance s conforms with K with the partial type $[\![K_\triangleleft \Rightarrow K_\triangleright]\!]$. We know that the locations of the required ports are $\{\,\mathit{l}_i\ ^{i \in 1..n}\} = \{\mathrm{select}_{S'}(s, \ell) \mid (\ell : \tau) \in K_\triangleleft\}$ and that $\mathsf{nil}(S') \subseteq \{\mathrm{select}_{S'}(s, \ell) \mid (\ell : \tau) \in K_\triangleleft\}$.

Γ_{n+1} types S_{n+1} and the locations in $s_\triangleleft = \{\ell_i = \mathit{l}_i\ ^{i \in 1..n}\}$ are in S_n. From the conformance of s with K, Definition 3.23, and Rule (Val Record) we know that $\Gamma_{n+1} \vdash s : \{\!| \ldots, \ell^p_j : \sigma_j\ ^{j \in 1..m} |\!\}$ where $K_\triangleright = \{\ell^p_j : \sigma_j\ ^{j \in 1..m}\}$ and by Rule (Val Interface), we have a $\Gamma' = \Gamma_{n+1}, \mathit{l} : K_\triangleright$, extending Γ, and typing $S_{n+1}[\,\mathit{l} \mapsto s][\,\mathit{l}_i \mapsto v_i\ ^{i \in 1..n}]$. We finally have that l is a location that `refers-to` an object, which is a record, such that $\Gamma' \vdash \iota : K_{\triangleright}$ with $K_{\triangleright} = \{\ell_j^p : \sigma_j\ ^{j\in 1..m}\}$. Finally, since $v_i \neq \mathsf{nil}$ for all $i \in 1..n$ we have that $\mathsf{nil}(S_{n+1}[\iota \mapsto s][\iota_j {\mapsto} v_j\ ^{j\in 1..m}]) = \emptyset$ with s being the newly created instance. Notice that ι and $S_{n+1}[\iota \mapsto s][\iota_j {\mapsto} v_j\ ^{j\in 1..m}]$ are record-based.

***Case:* (Eval Reconfig)**

If the last rule used is (Eval Reconfig) then we must have

a) $\mathsf{reconfig}\ x = e_1[e_2]\ \mathsf{with}\ \ell_i := f_i\ ^{i\in 1..n}\ \ \mathsf{in}\ e_3\ \mathsf{else}\ e_4; S \downarrow v; S'''$

and b) $\Gamma \vdash \mathsf{reconfig}\ x = e_1[e_2]\ \mathsf{with}\ \ell_i := e_i'\ ^{i\in 1..n}\ \mathsf{in}\ e_3\ \mathsf{else}\ e_4 : \delta$.

From b), by Rule (Val Reconfig), we have that $\Gamma \vdash e_1 : K \Longrightarrow K'$ and from a), by Rule (Eval Reconfig), we have $e_1; S \downarrow \mathsf{conf}(\tau', c); S'$. By induction hypothesis, we conclude that there is a Γ' that extends Γ and types S' such that $\Gamma' \vdash \mathsf{conf}(\tau', c) : K \Longrightarrow K'$, which, by Rule (Val Configurator Value), leads to $\tau' = K \Longrightarrow K'$ and $\Gamma' \vdash c : \tau'$. From b), by Rule (Val Reconfig), we also have that $\Gamma \vdash e_2 : \tau$. By Lemma 4.10 (weakening) we obtain $\Gamma' \vdash e_2 : \tau$, and $e_2; S' \downarrow \iota; S_0$ with $S_0(\iota) = s$. By induction hypothesis we conclude that there is a Γ'' which extends Γ' and types S_0 such that $\Gamma'' \vdash \iota : \tau$.

Again, from a), by Rule (Eval Reconfig), we now know that $s\,/\!/\,K$ and therefore, by Lemma 4.9, we know that s conforms with K and should have the partial linked object type $[\![\sigma \Rightarrow \tau]\!]$ for some object type σ.

The evaluation follows on the plug assigment expressions. By iterating Lemma 4.10 (weakening) and the induction hypothesis for all the expressions e_i, typed by $\Gamma_i \vdash e_i : \sigma_i$, we obtain that for each i there is a Γ_i that extends Γ_{i-1}, with $\Gamma_0 = \Gamma''$ typing each S_i such that $\Gamma_i \vdash v_i : \sigma_i$.

Then, we have that $s; c; S_n \Downarrow s'; S''$. By induction hypothesis on the second part of the lemma (the application of a configurator) we have that there is a Γ''' which extends Γ'' and that the resulting instance, s', conforms with K' and has the partial type $[\![K'_{\triangleleft} \oplus \sigma \Rightarrow K'_{\triangleright} \oplus \tau]\!]$ and its object type is $K'_{\triangleright} \oplus \tau$.

Let $\Gamma'''' = \Gamma''', \iota' : (\tau \oplus K'_{\triangleright})$. Again, from b), by Rule (Val Reconfig) and Lemma 4.10 (weakening), we obtain $\Gamma'''', x : (\tau \oplus K'_{\triangleright}) \vdash e_3 : \delta$ and by Lemma 4.11 (substitution) with the side condition $\Gamma'''' \vdash \iota' : (\tau \oplus K'_{\triangleright})$. We conclude c) $\Gamma'''' \vdash e_3[x \leftarrow \iota'] : \delta$. So, from a), by Rule (Eval Reconfig), we have $e_3[x \leftarrow \iota']; S_{n+1}[\iota' \mapsto s'][\iota_i {\mapsto} v_i\ ^{i\in 1..n}] \downarrow v; S''''$, and with c), by induction hypothesis, we obtain the final result that there is a Γ'''' such that $\Gamma'''' \vdash v : \delta$ with Γ'''' extending Γ''' types S''''.

Finally, the occurrence of nil values in the heap follows the reasoning used in the instantiation process. In the first case, all nil values introduced by the new required ports are replaced by non-nil values, so we prove that $\mathsf{nil}(S'''') = \emptyset$. The new values introduced in the heap are all record-based (by induction hypothesis) and so it is the result of evaluating e_3.

***Case:* (Eval Reconfig Else)**

The second evaluation possibility for a reconfiguration follows without applying the configurator to the instance and therefore the typing of the resulting value v results by induction hypothesis on the typing derivation of e_4.

When composition operations are involved the reasoning is also similar, but handling different value forms and using Rule (Val Configurator Value) to correctly type the results.

***Case:* (Eval Requires)**$^\rho$

The typing hypothesis for this case is $\Gamma \vdash (\mathsf{requires}\ \ell : I) : \sigma$ with $\sigma = \emptyset \Longrightarrow \{\ell \bullet I, \ell \lhd I\}$ where the expression $\mathsf{requires}\ \ell : \tau$ evaluates to a configurator. By Rule (Val Configurator Value)$^\rho$, we have $\Gamma' \vdash \mathsf{conf}(\sigma, \mathsf{requires}\ \ell : I) : \sigma$ with $\Gamma' = \Gamma$ and $S' = S$ with $\mathsf{nil}(S') = \emptyset$.

***Case:* (Eval Provides)**$^\rho$

The typing hypothesis for this case is $\Gamma \vdash (\mathsf{provides}\ \ell : \tau) : \sigma$ with $\sigma = \emptyset \Longrightarrow \{\ell \circ \tau, \ell \rhd \tau\}$ where the expression $\mathsf{requires}\ \ell : \tau$ evaluates to a configurator. By Rule (Val Configurator Value)$^\rho$, we have $\Gamma \vdash \mathsf{conf}(\sigma, \mathsf{provides}\ \ell : \tau) : \sigma$ with $\Gamma' = \Gamma$ and $S' = S$ with $\mathsf{nil}(S') = \emptyset$.

***Case:* (Eval Plug)**$^\rho$

The typing hypothesis for this case is $\Gamma \vdash \mathsf{plug}\ (\pi_1 : \tau)\ \mathsf{into}\ (\pi_2 : \tau) : \sigma$ with $\sigma = (\{\pi_2 \circ \tau, \pi_1 \bullet \tau\} \Longrightarrow \{\pi_1 \bullet \tau\})$ where the expression $\mathsf{plug}\ (\pi_1 : \tau)\ \mathsf{into}\ (\pi_2 : \tau)$ evaluates to a configurator. By Rule (Val Configurator Value)$^\rho$, we have $\Gamma \vdash \mathsf{conf}(\sigma, \mathsf{plug}\ (\pi_1 : \tau)\ \mathsf{into}\ (\pi_2 : \tau)) : \sigma$ with $\Gamma' = \Gamma$ and $S' = S$ with $\mathsf{nil}(S') = \emptyset$.

***Case:* (Eval Method Block)**$^\rho$

The typing hypothesis for this case is $\Gamma \vdash x_K[\ell_i : \tau_i = e_i\ ^{i \in 1..n}] : \sigma$ with $\sigma = K \Longrightarrow K, \{x \bullet \{\ell_i : \tau_i\ ^{i \in 1..n}\}\}$ where the expression $x_K[\ell_i : \tau_i = e_i\ ^{i \in 1..n}]$ evaluates to a configurator. By Rule (Val Configurator Value)$^\rho$, we have $\Gamma \vdash \mathsf{conf}(\sigma, x_K[\ell_i : \tau_i = e_i\ ^{i \in 1..n}]) : \sigma$ with $\Gamma' = \Gamma$ and $S' = S$ with $\mathsf{nil}(S') = \emptyset$.

***Case:* (Eval Uses)**$^\rho$

The evaluation of this expression produces a configurator where e is replaced by its resulting value, $x[e : \tau \Rightarrow \sigma]; S \downarrow \mathsf{conf}(\delta, x[v : \tau \Rightarrow \sigma]); S'$. The typing judgment implies $\Gamma \vdash x[e : \tau \Rightarrow \sigma] : \emptyset \Longrightarrow K$ with $\tau = \{\ell^r_i : \tau_i\ ^{i \in 1..n}\}, \sigma = \{\ell^p_j : \sigma_j\ ^{j \in 1..m}\}$ and $K = \{x \bullet \sigma, x.\ell^r_i \circ \tau_i\ ^{i \in 1..n}, x.\ell^p_j \bullet \sigma_j\ ^{j \in 1..m}\}$.

We have that $|\Gamma| \vdash e : \tau \Rightarrow \sigma$ and $e; S \downarrow v; S'$, which by induction hypothesis on the first part of the lemma implies that there is a $|\Gamma'|$ extending $|\Gamma|$ such that $|\Gamma'| \vdash v : \tau \Rightarrow \sigma$ and $\mathsf{nil}(S') = \emptyset$.

We conclude, by Rule (Val Configurator Value)$^\rho$, that $|\Gamma'| \vdash \mathsf{conf}(\delta, x[v : \tau]) : \emptyset \Longrightarrow K$ with $\delta = \emptyset \Longrightarrow K$ and by Lemma 4.10 (weakening) that there is a Γ' extending Γ and $\Gamma' \vdash \mathsf{conf}(\delta, x[v : \tau]) : \emptyset \Longrightarrow K$.

***Case:* (Eval Sequence)**

By induction hypothesis we reach the conclusion that there is a Γ' extending Γ and typing S' such that $\Gamma' \vdash \mathsf{conf}\sigma, c_1) : \sigma$ with $\sigma = (K \Longrightarrow K', K_c)$ and $\mathsf{nil}(S') = \emptyset$. Also by induction hypothesis, there is Γ'' extending both Γ' and Γ and typing S'' such that $\Gamma'' \vdash \mathsf{conf}(\sigma', c_2) : \sigma'$ with $\sigma' = (K_c, K'' \Longrightarrow K''')$ and $\mathsf{nil}(S'') = \emptyset$. By Rule (Val Configurator Value) we conclude that $\Gamma'' \vdash \mathsf{conf}(\sigma'', (c_1; c_2)) : \sigma''$ with $\sigma'' = K, K'' \Longrightarrow K', K'''$.

We follow by proving that well-typed expressions never evaluate to wrong.

Case: **(Wrong Call)**

For an application to be well-typed we must have $\Gamma \vdash e_1(e_2) : \sigma$. Then, by inspection of the type system, we conclude that the only typing rule that may derive this judgment is Rule (Val Application), and therefore we have a) $\Gamma \vdash e_1 : \tau \to \sigma$ and b) $\Gamma \vdash e_2 : \sigma$. We also have that c) $e_1; S \downarrow v; S'$ which by induction hypothesis lets us conclude that there is a Γ' typing S' such that $\Gamma \vdash v : \tau \to \sigma$ and v is an abstraction. Therefore if an application expression is well-typed, the Rule (Wrong Call) is never applicable.

Case: **(Wrong Assign)**

If $\Gamma \vdash e_1.\ell := e_2 : \tau$ then by Rule (Eval Assign) we know that a) $\Gamma \vdash e_1 : \{\!|\, \ldots, \ell : \tau, \ldots \,|\!\}$. The evaluation hypothesis has the premise b) $e_1; S \downarrow v; S'$. Taking a) and b), by induction hypothesis we know that there is Γ' typing S' and extending Γ such that $\Gamma \vdash v : \{\!|\, \ldots, \ell : \tau, \ldots \,|\!\}$ and v must be a location leading to a record. Thus Rule (Wrong Assign) is never applicable to well-typed assignment expressions.

Case: **(Wrong Assign 2)**

Following the reasoning from the previous case, we have $\Gamma \vdash \imath : \{\!|\, \ldots, \ell : \tau, \ldots \,|\!\}$ and that $\imath$ is a location leading to a record. By Definition 2.21 we know that $\Gamma \vdash S(\imath) : \{\!|\, \ldots, \ell : \tau, \ldots \,|\!\}$ which by Rule (Val Record) must contain the label ℓ and therefore Rule (Wrong Assign 2) is also never applicable.

Case: **(Wrong Select)**. Similar.

Case: **(Wrong Select 2)**. Similar.

Case: **(Wrong Compose)**

If $\Gamma \vdash \mathsf{compose}\ e : K_\triangleleft \Rightarrow K_\triangleright$ then we know that $\Gamma \vdash e : \emptyset \Longrightarrow K$ by induction hypothesis we have a Γ' extends Γ and types S' and that v is a value such that $\Gamma \vdash v : \emptyset \Longrightarrow K$. This value can only be a configurator value. So wrong is never issued by Rule (Wrong Compose).

Case: **(Wrong New)**

If $\Gamma \vdash \mathsf{new}\ e\ \mathsf{with}\ \ell^r_j := e_j\ ^{j \in 1..m} : \sigma$, by Rule (Val New), we have that $\Gamma \vdash e : \tau \Rightarrow \sigma$ and that

$e; S \downarrow v; S'$. By induction hypothesis we obtain that there is a Γ' extending Γ and typing S' such that v is a component value (other possible values have different types). This is contradicts the application of (Wrong New).

***Case:* (Wrong Sequence)**

If $\Gamma \vdash e_1; e_2 : K, K'' \Longrightarrow K', K'''$ then $\Gamma \vdash e_1 : K \Longrightarrow K', K_c$ and, as $e_1; S \downarrow v; S$ we know, by induction hypothesis, that there is Γ' extending Γ and typing S' such that v is a configurator value with $\Gamma' \vdash v : K \Longrightarrow K', K_c$. Once again, we find a contradiction.

***Case:* (Wrong Sequence 2)**. Similar.

***Case:* (Wrong Reconfig)**

By induction hypothesis on the premises typing and evaluating e_1, we know that there is a Γ' that extends Γ and types S' such that e_1 evaluates to a value v typed by $\Gamma' \vdash v : K \Longrightarrow K'$ that is either an abstraction, a component, a configurator, or a location that *refers-to* a record. By our typing relation we conclude that this is a configurator $\mathsf{conf}(K \Longrightarrow K', c)$. So, Rule (Wrong Reconfig) is never applicable.

***Case:* (Wrong Reconfig 2)** and (Wrong Reconfig 3)

From the previous case we know that the value to which e_1 evaluates is a configurator, and if e_2 is typed by an interface, and $e_2; S' \downarrow v; S_0$. by induction hypothesis we know that it must be a location that is either *undefined* or *refers-to* a record. So, Rules (Wrong Reconfig 2) and (Wrong Reconfig 3) are never applied.

The second part of the lemma states that the application of a configurator to an instance causes an effect consistent with the type of the operation. This proof is done by induction on the height of the derivations and by case analysis of the last application rule used. We verify that the type information in the instances is, at any time, sound with the global typing of the heap. We use the first part of the lemma when necessary.

***Case:* (App Provides)$^\rho$**

In this case we have an expression typed $\Gamma \vdash (\mathsf{provides}\ \ell : \tau) : \emptyset \Longrightarrow \{\ell \circ \tau, \ell \triangleright \tau\}$ and an instance $s = (r, e, p)_\Pi$ that conforms with $\emptyset$. The resulting instance, $s' = (r, e, p \oplus \{\ell = \imath\})_{\Pi, \imath:\tau}$, trivially conforms, by Definition 3.23, with the resource set $\{\ell \circ \tau, \ell \triangleright \tau\}$ and the typing environment $\Gamma' = \Gamma, \imath : \tau$ which also types the heap $S' = S[\imath \mapsto \mathsf{nil}]$. The resulting value is well-typed according to Rule (Val Object).

Since the demanded resources in the configurator type is empty we have that $\mathsf{nil}(S) \subseteq X$ for some X. By Rule (App Provides)$^\rho$ we know that $\mathrm{select}_{S'}(s', \ell) = \imath$ and that $S'(\imath) = \mathsf{nil}$. More, $\imath$ is the only new location in S' with relation to S and $K'_\triangleleft \cup K'_\circ = \{\ell : \tau\}$. So, $\{\mathrm{select}_{S'}(s', \ell) \mid (\ell :$

$\tau) \in K'_{\lhd} \cup K'_{\circ}\} = \{\imath\}$ and therefore $\mathsf{nil}(S') \subseteq \{\imath\} \cup X$ which corresponds to the expected results. With relation to the partially linked object type, we have that if the initial type of s is $[\![\tau \Rightarrow \sigma]\!]$ then the partial type of s' is $[\![\tau \Rightarrow \sigma \oplus \{\ell : \tau\}]\!]$.

***Case:* (App Requires)$^{\rho}$**

In this case we have the typing judgment $\Gamma \vdash (\mathsf{requires}\ \ell : \tau) : \emptyset \Longrightarrow \{\ell \bullet \tau, \ell \lhd \tau\}$ as hypothesis and an instance $s = (r, e, p)_{\Pi}$ that conforms with $\emptyset$. The resulting instance, $s' = (r \oplus \{\ell = \imath\}, e, p)_{\Pi, \imath : \tau}$, trivially conforms, by Definition 3.23, with the resource set $\{\ell \bullet \tau, \ell \lhd \tau\}$ and the typing environment $\Gamma' = \Gamma, \imath : \tau$, which also types the heap $S' = S[\imath \mapsto \mathsf{nil}]$. The resulting value is well-typed according to Rule (Val Object).

Since the demanded resources in the configurator type is empty we have that $\mathsf{nil}(S) \subseteq X$ for some X. By Rule (App Requires)$^{\rho}$ we know that $\mathrm{select}_{S'}(s', \ell) = \imath$ and that $S'(\imath) = \mathsf{nil}$. More, $\imath$ is the only new location in S' with relation to S and $K'_{\lhd} \cup K'_{\circ} = \{\ell : \tau\}$. So, $\{\mathrm{select}_{S'}(s', \ell) \mid (\ell : \tau) \in K'_{\lhd} \cup K'_{\circ}\} = \{\imath\}$ and therefore $\mathsf{nil}(S') \subseteq \{\imath\} \cup X$ which corresponds to the expected results. With relation to the partially linked object type, we have that if the initial type of s is $[\![\tau \Rightarrow \sigma]\!]$ then the partial type of s' is $[\![\tau \oplus \{\ell : \tau\} \Rightarrow \sigma]\!]$.

***Case:* (App Uses)$^{\rho}$**

If the last evaluation rule is (App Uses)$^{\rho}$ then we have a) $s; x[v : \tau]; S \Downarrow s'; S'$ and also
b) $\Gamma \vdash x[v : \tau] : \emptyset \Longrightarrow K$ with $K = \{x \bullet \{\ell^p_j : \sigma'_j\ {}^{j \in 1..m}\}, x.\ell^r_i \circ \tau'_i\ {}^{i \in 1..n}, x.\ell^p_j \bullet \sigma'_j\ {}^{j \in 1..m}\}$ where
$\tau = \{\ell^r_i : \tau_i\ {}^{i \in 1..k}\} \Rightarrow \{\ell^p_j : \sigma_j\ {}^{j \in 1..m}\}$.

From b), by Rule (Comp Uses), we have that $|\Gamma| \vdash v : \tau$, and since the only kind of values typed in this way are components, we also have that $v = \mathsf{comp}(c)$ with $|\Gamma| \vdash \mathsf{comp}(c) : \tau$. By Rule (Val Composition Value) we known that $|\Gamma| \vdash \mathsf{compose}\ c : \tau$ and from Rule (Val Compose) we know c) $|\Gamma| \vdash c : \emptyset \Longrightarrow K'$ with $K'_{\circ} = \emptyset$ and $\tau = (K'_{\lhd} \Rightarrow K'_{\rhd})$.

From a), by Rule (App Uses)$^{\rho}$, we know that $(\mathsf{new}\ v); S \downarrow \imath; S'$. Since v evaluates to itself causing no changes to the heap, Rule (Eval Value), and there are no plug assignments to be considered, the evaluation of $\mathsf{new}\ v$ is supported, by Rule (Eval New), on d) $\mathbf{0}; c; S \Downarrow s''; S''$ with $S' = S''[\imath \mapsto s'']$.

Since the set of demanded resources in the type assigned in b) ($\emptyset \Longrightarrow K$) is empty, we know, from the conditions of the lemma, that there is a set X such that $\mathsf{nil}(S) \subseteq X$. From c) and d), by induction hypothesis on the second case of the lemma, we have that there is a $|\Gamma'|$ that types S'' and that the resulting value, s'', conforms with K' with the partially linked object type $[\![K'_{\lhd} \Longrightarrow K'_{\rhd}]\!]$ with relation to $|\Gamma'|$. Remember that the initial instance is $\mathbf{0}$ and therefore its partial object type is $[\![\{\} \Rightarrow \{\}]\!]$.

The instance resulting from applying $x[v : \tau]$ to s is therefore $s' = (r, e \oplus \{x = \imath\}, p)_{\Pi, \imath : K'_{\rhd}}$, see (App Uses), and the resulting heap is S'. They are both well typed with relation to $|\Gamma''| =$

$|\Gamma'|, \iota : K'_{\triangleright}$ and by weakening with relation to Γ'' (by adding the elements taken from Γ to produce $|\Gamma|$).

More, the instance $(r, e \oplus \{x = \iota\}, p)_{\Pi,\iota:K'_{\triangleright}}$, according to Definition 3.23, conforms with K. The induction hypothesis also tells us that $\mathsf{nil}(S'') \subseteq \{\mathrm{select}_{S''}(s,\ell) \mid (\ell : \tau) \in (\ell^r_i : \tau_i)^{i\in 1..n}\} \cup X$ with $K'_{\triangleleft} = \{\ell^r_i : \tau_i{}^{i\in 1..k}\}$. Which, in the involving composition context, can be written by dereferrencing x, $\mathsf{nil}(S') \subseteq \{\mathrm{select}_{S''}(s', \pi) \mid (\pi : \tau) \in K_{\circ}\} \cup X$ where $K_{\circ} = \{x.\ell^r_i : \tau_i{}^{i\in 1..n}\}$.

The partial type of the resulting object remains unchanged since no required or provided ports are added to the instance.

***Case:* (App Method Block)**

If $\Gamma \vdash x_K[\ell_i : \tau_i = v_i{}^{i\in 1..n}] : K \implies K, \{x \bullet \{\ell_i : \tau_i{}^{i\in 1..n}\}\}$, then by Rule (Comp Method Block), we know that for all $i \in 1..n$, $|\Gamma|, x : \{\!|\ell_i : \tau_i{}^{i\in 1..n}|\!\}, K_{\bullet} \vdash v_i : \tau_i$. Since $(r,e,p)_{\Pi}$ conforms with K and $v_i[(r,e,p)_{\Pi}][x \leftarrow \iota]$ denotes the substitution of the available names in the current instance $(r,e,p)_{\Pi}$ by its locations, which are typed in Γ, by Lemma 4.10 (weakening) and Lemma 4.11 (substitution), we obtain $\Gamma, x : \{\!|\ell_i : \tau_i{}^{i\in 1..n}|\!\} \vdash v_i[(r,e,p)_{\Pi}] : \tau_i$ for all $i \in 1..n$ ($|\Gamma|$ correctly types the instance). Let $\Gamma' = \Gamma, \iota : \{\!|\ell_i : \tau_i{}^{i\in 1..n}|\!\}$ and by Lemma 4.10 (weakening) and 4.11 (substitution) we have that $\Gamma' \vdash v_i[(r,e,p)][x \leftarrow \iota] : \tau_i$ for all $i \in 1..n$. So, we have that $\Gamma', \iota_i : \tau_i{}^{i\in 1..n}$ types the resulting heap $S' = S[\iota \mapsto \{\ell_i = \iota_i{}^{i\in 1..n}\}][\iota_i \mapsto v_i[(r,e,p)][x \leftarrow \iota]{}^{i\in 1..n}]$.

Since $(r,e,p)_{\Pi}$ conforms with K this makes the instance, $(r, e \oplus \{x = \iota\}, p)$, conform with $K' = K, \{x \bullet \{\ell_i : \tau_i{}^{i\in 1..n}\}\}$. No references are created in the heap that lead to nil, hence $\mathsf{nil}(S') = \mathsf{nil}(S)$, and by considering the set X such that $\mathsf{nil}(S) \subseteq \{\mathrm{select}_S(s,\ell) \mid (\ell : \tau) \in K_{\triangleleft} \cup K_{\circ}\} \cup X$ we have that $\mathsf{nil}(S') \subseteq \{\mathrm{select}_S(s,\ell) \mid (\ell : \tau) \in K'_{\triangleleft} \cup K'_{\circ}\} \cup X$ which is the same set. As in the previous case, the partial object type remains unchanged.

***Case:* (App Plug)**

If $\Gamma \vdash \mathsf{plug}\ (\pi_1 : \tau)\ \mathsf{into}\ (\pi_2 : \tau) : (\{\pi_2 \circ \tau, \pi_1 \bullet \tau\} \implies \{\pi_1 \bullet \tau\})$ only the unsatisfied inner requirements change, So $K'_{\circ} \subseteq K_{\circ}$, with $K = \{\pi_2 \circ \tau, \pi_1 \bullet \tau\}$ and $K' = \{\pi_1 \bullet \tau\}$. However, the resulting instance is the same and by Definition 3.23, s conforms with K'. On the other hand, the heap is changed to make the connection between the source and the target of the plug operation. From the lemma's hypothesis we know that $\mathsf{nil}(S) \subseteq \{\mathrm{select}_S(s,\ell) \mid (\ell : \tau) \in K_{\triangleleft} \cup K_{\circ}\} \cup X$. $S(\mathrm{select}_S(s,\pi_2)) = \mathsf{nil}$, hence $\mathsf{nil}(S[\mathrm{select}_S(s,\pi_2) \mapsto \mathrm{select}_S(s,\pi_1)]) \subseteq \{\mathrm{select}_S(s,\ell) \mid (\ell : \tau) \in K'_{\triangleleft} \cup K'_{\circ}\} \cup X$.

***Case:* (App Sequence)**

We first prove that the resulting instance conforms with the type. The typing judgement $\Delta \vdash e_1; e_2 : K, K'' \implies K', K'''$ is supported by the premises $\Delta \vdash e_1 : K \implies K', K_c$ and $\Delta \vdash e_2 : K'_c, K'' \implies K'''$. By induction hypothesis on the second case of the lemma, together with the application of e_1 to an instance s conforms with K we reach an instance s' which conforms with K', K_c. Since s' extends s it also conforms with K'' and by Lemma 4.10 (weakening) and induction hypothesis on the second case of the lemma together with the evaluation of e_2 we conclude that s'' conforms with K'''. Since it extends s and s' we conclude that it also conforms with K'.

Now concerning the nil values in the heap. In the case of $s; (c_1; c_2); S \Downarrow s''; S''$ with $\Delta \vdash (e_1; e_2) : K, K'' \Longrightarrow K', K'''$ we have that there is a set X such that $\mathsf{nil}(S) \subseteq \{\mathrm{select}_S(s, \ell) \mid (\ell : \tau) \in (K, K'')_{\triangleleft} \cup (K, K'')_{\circ}\} \cup X$. By rewriting this definition we obtain $\mathsf{nil}(S) \subseteq \{\mathrm{select}_S(s, \ell) \mid (\ell : \tau) \in K_{\triangleleft} \cup K_{\circ}\} \cup X'$ with $X' = \{\mathrm{select}_S(s, \ell) \mid (\ell : \tau) \in K''_{\triangleleft} \cup K''_{\circ}\} \cup X$. Given the application of the first premise $s; c_1; S \Downarrow s'; S'$ typed $\Delta \vdash e_1 : K \Longrightarrow K', K_c$, by induction hypothesis, we know that $\mathsf{nil}(S') \subseteq \{\mathrm{select}_S(s, \ell) \mid (\ell : \tau) \in (K', K_c)_{\triangleleft} \cup (K', K_c)_{\circ}\} \cup X'$. Which is the same as $\mathsf{nil}(S') \subseteq \{\mathrm{select}_S(s, \ell) \mid (\ell : \tau) \in K'_{\triangleleft} \cup K'_{\circ}\} \cup \{\mathrm{select}_S(s, \ell) \mid (\ell : \tau) \in K_{c\triangleleft} \cup K_{c\circ}\} \cup \{\mathrm{select}_S(s, \ell) \mid (\ell : \tau) \in K''_{\triangleleft} \cup K''_{\circ}\} \cup X$. Again, by rearranging the definition we have $\mathsf{nil}(S') \subseteq \{\mathrm{select}_S(s, \ell) \mid (\ell : \tau) \in K'', K_{c\triangleleft} \cup K'', K_{c\circ}\} \cup X''$ with $X'' = \{\mathrm{select}_S(s, \ell) \mid (\ell : \tau) \in K'_{\triangleleft} \cup K'_{\circ}\} \cup X$. Given the second premise evaluated by $s; c_1; S \Downarrow s'; S'$ and typed by $\Delta \vdash e_2 : K_c, K'' \Longrightarrow K'''$, by induction hypothesis we know that $\mathsf{nil}(S'') \subseteq \{\mathrm{select}_S(s, \ell) \mid (\ell : \tau) \in K', K'''_{\triangleleft} \cup K', K'''_{\circ}\} \cup X''$. As the locations in X'' corresponding to the labels in K' are already in this final set we conclude that $\mathsf{nil}(S'') \subseteq \{\mathrm{select}_S(s, \ell) \mid (\ell : \tau) \in K', K'''_{\triangleleft} \cup K', K'''_{\circ}\} \cup X$. □

A.3 Chapter 4

(This is a repetition of Lemma 5.3, defined in page 117.)

Lemma 5.3. *For all $\Delta, \Delta' \in \mathcal{D}$ and types $\delta \in \mathcal{T}_\lambda$, If $\Delta \vdash \tau$ ok then $\Delta \vdash \diamond$.*

Proof. We prove this by induction on the height of the derivations and by case analysis on the last rule used.

***Case:* (Type Top) and (Type TVar)**

In this case the conclusion is immediate from the premises of the rule.

***Case:* (Type Fun) and (Type All)**

By induction on one of the premises.

***Case:* (Type Rec)**

$\Delta \vdash \mu X.\tau$ ok has as premise $\Delta, X \leq \top \vdash \tau$ ok which by induction hypothesis implies that $\Delta, X \leq \top \vdash \diamond$ which is only true if $\Delta \vdash \top$ ok and consequently if $\Delta \vdash \diamond$. □

(This is a repetition of Lemma 5.4, defined in page 117.)

Lemma 5.4 (Weakening of typing environments)**.** *For all $\Delta, \Delta' \in \mathcal{D}$ and types $\delta \in \mathcal{T}_\lambda$,*

1. *if $\Delta, \Delta' \vdash \diamond$, $X \notin Dom(\Delta, \Delta')$, and $\Delta \vdash \delta$ ok then $\Delta, X \leq \delta, \Delta' \vdash \diamond$.*

2. *if $\Delta, \Delta' \vdash \tau$ ok, $X \notin Dom(\Delta, \Delta')$, and $\Delta \vdash \delta$ ok then $\Delta, X \leq \delta, \Delta' \vdash \tau$ ok.*

Proof. For the first case we prove by induction on the height of the derivations and in the two possible cases for $\Delta' = \phi$.

Case: $\Delta' = \phi$

In this case we have $\Delta \vdash \delta$ ok and by Rule (Env TVar) we conclude that $\Delta, X \leq \delta \vdash \diamond$.

Case: $\Delta' \neq \phi$

In this case we have that $\Delta' = \Delta'', Y \leq \tau$ and we know that $\Delta, \Delta'' \vdash \tau$ ok and that $Y \notin Dom(\Delta, \Delta'')$. By induction hypothesis we know that $\Delta, X \leq \delta, \Delta'' \vdash \tau$ ok. So, we conclude that $\Delta, X \leq \delta, \Delta' \vdash \diamond$.

For the second case we apply the induction hypothesis in the cases of τ and applying the first case in the case of $\top$.

Case: $\tau = \top$

We have $\Delta, \Delta' \vdash \top$ ok if $\Delta, \Delta' \vdash \diamond$ and by induction hypothesis we conclude that $\Delta, X \leq \delta, \Delta' \vdash \diamond$ and by (Type Top) we conclude that $\Delta, X \leq \delta, \Delta' \vdash \top$ ok.

Case: $\tau \neq \top$

Follows by induction hypothesis on the premises of the rule applied. □

(This is a repetition of Lemma 5.5, defined in page 118.)

Lemma 5.5 (Substitution)**.** *For all $\Delta, \Delta' \in \mathcal{D}$ and types $\delta \in \mathcal{T}_\lambda$,*

1. *If $\Delta, X \leq \delta, \Delta' \vdash \diamond$ and $\Delta \vdash \delta'$ ok then $\Delta, \Delta'[X \leftarrow \delta'] \vdash \diamond$.*

2. *If $\Delta, X \leq \delta, \Delta' \vdash \tau$ ok and $\Delta \vdash \delta'$ ok then $\Delta, \Delta'[X \leftarrow \delta'] \vdash \tau[X \leftarrow \delta']$ ok.*

Proof. We prove this lemma by mutual induction of the two cases. The first case is proven by induction on the height of the derivations and in two possible cases for $\Delta' = \phi$.

Case: $\Delta' = \phi$

In this case we have $\Delta, X \leq \delta \vdash \diamond$ which must be derived from $\Delta \vdash \delta$ ok. By Lemma 5.3 we have that $\Delta \vdash \diamond$.

Case: $\Delta' \neq \phi$

In this case $\Delta, X \leq \delta, \Delta' \vdash \diamond$ and $\Delta' = \Delta'', Y \leq \delta'$ then it must be derived from (Env TVar) which is implied by $\Delta, X \leq \delta, \Delta'' \vdash \delta'$ ok. Then, by the second case of the lemma, we have that $\Delta, \Delta''[X \leftarrow \delta] \vdash \delta'[X \leftarrow \delta]$ which, by Rule (Env TVar) leads to $\Delta, \Delta'[X \leftarrow \delta] \vdash \diamond$.

We prove the second case of the lemma by induction on the height of the derivations and in the cases of the last rule used.

Case: **(Type Top) and (Type TVar)**

It follows from applying the first case of the lemma on the premises.

Case: **(Type Fun), (Type All), and (Type Rec)**

By applying the induction hypothesis on the premises. □

(This is a repetition of Lemma 5.6, defined in page 118.)

Lemma 5.6 (Variable Exchange). *For all $\Delta, \Delta' \in \mathcal{D}$ and types $\delta \in \mathcal{T}_\lambda$,*

1. *If $\Delta, X \leq \delta, \Delta', \Delta'' \vdash \diamond$ and $X \notin Dom(\Delta')$ then $\Delta, \Delta', X \leq \delta, \Delta'' \vdash \diamond$.*
2. *If $\Delta, X \leq \delta, \Delta', \Delta'' \vdash \tau \text{ ok}$ and $X \notin Dom(\Delta')$ then $\Delta, \Delta', X \leq \delta, \Delta'' \vdash \tau \text{ ok}$.*

Proof. For the first case, by iterating Rule (Env TVar) and Lemma 5.3 we have that $\Delta, X \leq \delta, \Delta' \vdash \diamond$. As $X \notin \Delta'$ then, by Lemma 5.5 we have $\Delta, \Delta' \vdash \text{ ok}$ and therefore $\Delta, \Delta', X \leq \delta \vdash \text{ ok}$. We can then reconstruct the typing environment to obtain $\Delta, \Delta', X \leq \delta, \Delta'' \vdash \diamond$.

The second case is proven by induction on height of the derivations by applying the induction hypothesis on the premises of the last rule used. □

(This is a repetition of Proposition 7, defined in page 119.)

Proposition 5.9 (Monotonicity of S). *For all $\mathfrak{R}, \mathfrak{R}' \in \mathcal{P}(\mathcal{J})$, $\mathfrak{R} \subseteq \mathfrak{R}' \Rightarrow S(\mathfrak{R}) \subseteq S(\mathfrak{R}')$.*

Proof. The proof is done by case analysis on $(\Delta, \tau, \sigma) \in S(\mathfrak{R})$:

Case: $\tau \equiv \sigma$ **(Sub Equal) and** $\sigma \equiv \top$ **(Sub Top).**

In this case we have that $\tau \in \mathcal{T}_\lambda$ and $\Delta \vdash \tau \text{ ok}$. By the definition of S we know that for any other relation $\mathfrak{R}'' \in \mathcal{P}(\mathcal{J})$, $(\Delta, \tau, \sigma) \in S(R)$. In particular this is true for $\mathfrak{R}'$.

Case: $\tau \equiv X$ **(Sub TransVar).**

There is a type $\delta \in \mathcal{T}_\lambda$ such that $(\Delta, \delta, \sigma) \in \mathfrak{R}$ with $X \leq \delta \in \Delta$. As $\mathfrak{R} \subseteq \mathfrak{R}'$ then $(\Delta, \delta, \sigma) \in \mathfrak{R}'$ and by the definition of S we have that $(\Delta, \delta, \sigma) \in S(\mathfrak{R}')$.

Case: $\tau \equiv \tau \to \sigma$ **and** $\sigma \equiv \tau' \to \sigma'$ **(Sub Fun)**

The support for this case is $(\Delta; \tau'; \tau)$ and $(\Delta; \sigma; \sigma')$ both in $\mathfrak{R}$ and $\mathfrak{R}'$. So, by definition of S we have that $(\Delta; \tau; \sigma) \in \mathfrak{R}'$.

Case: $\tau \equiv \forall_{X \leq \tau} \tau'$ **and** $\tau \equiv \forall_{X \leq \sigma} \sigma'$ **(Sub All)**. Similar.

Case: $\sigma \equiv \mu X.\sigma'$ **(Sub RecR)**

In this case we know that $(\Delta, \tau, \sigma'[X \leftarrow \mu X.\sigma']) \in \mathfrak{R}$ and also in $\mathfrak{R}'$. Thus, $(\Delta, \tau, \mu X.\sigma') \in S(\mathfrak{R}')$.

Case: $\tau \equiv \mu X.\tau'$ **(Sub RecL)**. Similar. □

(This is a repetition of Lemma 5.11, defined in page 119.)

Lemma 5.11 (Weakening)**.** *For all typing environments $\Delta, \Delta' \in \mathcal{D}$, and types $\tau, \sigma, \delta \in \mathcal{T}_\lambda$, if $\Delta, \Delta' \vdash \tau \leq \sigma$, $X \notin Dom(\Delta, \Delta')$, and $\Delta \vdash \delta$ ok then $\Delta, X \leq \delta, \Delta' \vdash \tau \leq \sigma$.*

Proof. We first define the set of all judgments that result from weakening judgments of νS as:

$$\mathfrak{W} \triangleq \{(\Delta, X \leq \delta, \Delta'; \tau; \sigma) \mid (\Delta, \Delta'; \tau; \sigma) \in \nu S,\ X \notin Dom(\Delta, \Delta') \text{ and } \Delta \vdash \delta \text{ ok}\}$$

and then prove that $\mathfrak{W} \subseteq \nu S$ by the coinduction principle.

Let $(\Delta, X \leq \delta, \Delta'; \tau; \sigma)$ be a tuple of $\mathfrak{W}$. There is $(\Delta, \Delta'; \tau; \sigma) \in \nu S$ such that $X \notin Dom(\Delta, \Delta')$ and $\Delta \vdash \delta$ ok. We now analyze the different cases of S that support $(\Delta, \Delta'; \tau; \sigma) \in \nu S$.

Case: $\tau \equiv \sigma$ **(Sub Equal)**

In this case we have that $\Delta, \Delta' \vdash \tau$ ok. We know that $X \notin Dom(\Delta, \Delta')$, and $\Delta \vdash \delta$ ok. By lemma 5.4 we have that $\Delta, X \leq \delta, \Delta' \vdash \tau$ ok and therefore $(\Delta, X \leq \delta, \Delta'; \tau; \sigma) \in S(R)$ for any simulation R and in particular for $\mathfrak{W}$.

Case: $\sigma \equiv \top$ **(Sub Top)**. Similar.

Case: $\tau \equiv Y$ **(Sub TransVar)**

There is a type $\tau \in \mathcal{T}_\lambda$ such that $(\Delta, \Delta'; \tau; \sigma) \in \nu S$ with $Y \leq \tau \in \Delta, \Delta'$. By definition we have that $(\Delta, X \leq \delta, \Delta'; \tau; \sigma) \in \mathfrak{W}$ and by definition of S, $(\Delta, X \leq \delta, \Delta'; Y; \sigma) \in S(\mathfrak{W})$.

Case: $\sigma \equiv \mu Y.I$ **(Sub RecR)**

In this case we know that $(\Delta, \Delta'; \tau; I[Y \leftarrow \mu Y.I]) \in \nu S$ and so $(\Delta, X \leq \delta, \Delta'; \tau; I[Y \leftarrow \mu Y.I]) \in \mathfrak{W}$. Then $(\Delta, X \leq \delta, \Delta'; \tau; \sigma) \in S(\mathfrak{W})$.

Case: $\tau \equiv \mu Y.\tau$ **(Sub RecL)**. Similar.

Case: $\tau \equiv \tau' \to \tau''$ **(Sub Fun)**

By inspection of the definition of S we have $\tau \equiv \tau' \to \tau''$ and $\sigma \equiv \sigma' \to \sigma''$, and the tuple $(\Delta, \Delta'; \tau; \sigma) \in \nu S$ supported by the tuples $(\Delta, \Delta'; \sigma'; \tau')$, $(\Delta, \Delta'; \tau''; \sigma'') \in \nu S$. Thus, by definition $(\Delta, X \leq \delta, \Delta'; \sigma'; \tau')$, $(\Delta, X \leq \delta, \Delta'; \tau''; \sigma'') \in \mathfrak{W}$. By definition of S, $(\Delta, X \leq \delta, \Delta'; \tau; \sigma) \in S(\mathfrak{W})$.

Case: $\tau \equiv \forall_{X \leq \delta}\, \tau'$ **and** $\sigma \equiv \forall_{X \leq \delta}\, \sigma'$ **(Sub All)**

This tuple is supported by the tuple $(\Delta, \Delta', X \leq \delta; \tau; \tau') \in \nu S$ which by a reasoning similar to the one applied in the previous case, leads to the conclusion. □

(This is a repetition of Proposition 8, defined in page 120.)

Proposition 5.14 (νS is closed under exchange)**.** *For all $\Delta, \Delta', \Delta'' \in \mathcal{P}(\mathcal{J})$ and $\tau, \sigma, \delta \in \mathcal{T}$, If $(\Delta, X \leq \delta, \Delta', \Delta''; \tau; \sigma) \in \nu S$ and $X \notin FV(\Delta')$ then $(\Delta, \Delta', X \leq \delta, \Delta''; \tau; \sigma) \in \nu S$.*

Proof. We first define the auxiliary set $\mathfrak{P}$ by:

$$\begin{aligned} \mathfrak{P} \triangleq\ & \{(\emptyset, \tau, \sigma) | (\emptyset, \tau, \sigma) \in \nu S\} \\ \cup\ & \{(\Delta, \Delta', X \leq \delta, \Delta''; \tau; \sigma) \mid (\Delta, X \leq \delta, \Delta', \Delta''; \tau; \sigma) \in \nu S \text{ and } X \notin FV(\Delta')\}. \end{aligned}$$

Note that $\nu S \subseteq \mathfrak{P}$ and that all the tuples in νS that have empty typing environments are in $\mathfrak{P}$ by the first case of the definition. The remaining tuples of νS are in $\mathfrak{P}$ by the second case with $\Delta' = \emptyset$. We prove that $\mathfrak{P}$ is *S-consistent* and therefore by the coinduction principle that $\mathfrak{P} \subseteq \nu S$. $\mathfrak{P}$ is *S-consistent* if $\mathfrak{P} \subseteq S(\mathfrak{P})$. So, pick a tuple $(\Delta, \tau, \sigma) \in \mathfrak{P}$, if the tuple is supported by the first case of $\mathfrak{P}$'s definition then it is a member of νS and the possible cases of that happening are the ones that allow empty typing environments:

Case: $\tau = \sigma$ **(Sub Equal) and** $\sigma = \top$ **(Sub Top)**

$(\emptyset, \tau, \sigma) \in S(\mathfrak{P})$ by definition of S.

Case: $\sigma = \mu Y.\sigma'$ **(Sub RecR)**

In this case we know that if $(\emptyset, \tau, \mu Y.\sigma') \in \nu S$ it is because $(\emptyset, \tau, \sigma'[Y \leftarrow \mu Y.\sigma']) \in \nu S$ which, by definition, is also in $\mathfrak{P}$. Therefore $(\emptyset, \tau, \mu Y.\sigma') \in S(\mathfrak{P})$.

Case: $\tau = \mu Y.\tau'$ **(Sub RecL)**. Similar.

Case: $\tau = \forall_{X \leq \delta}\, \tau'$ **(Sub All)**

In this case we have $\sigma = \forall_{X \leq \delta}\, \sigma'$ and that $(\emptyset, \tau, \sigma) \in \nu S$ is supported by the tuple $(\emptyset, X \leq \delta; \tau'; \sigma') \in \nu S$. Which, by definition, is also an element of $\mathfrak{P}$ and hence $(\emptyset; \tau; \sigma) \in S(\mathfrak{P})$.

Case: $\tau = \tau' \to \tau''$ **(Sub Fun)**

As in the previous case, the tuples that support this case are also elements of $\mathfrak{P}$ and therefore the conclusion is also true.

The remaining case in the definition of S, (Sub TransVar), does not apply it demands a non-empty typing environment. On the other hand, if the tuple is supported by the second case of $\mathfrak{P}$'s definition then it is of the form $(\Delta, \Delta', X \leq \delta, \Delta''; \tau; \sigma)$ and there is a tuple $(\Delta, X \leq \delta, \Delta', \Delta''; \tau; \sigma) \in \nu S$ that supports it. We now analyse the cases by which $(\Delta, X \leq \delta, \Delta', \Delta''; \tau; \sigma)$ is in turn supported in νS.

Case: $\tau = \sigma$ **(Sub Equal)**

We know that $\Delta, X \leq \delta, \Delta', \Delta'' \vdash \tau\ \mathsf{ok}$, by Lemma 5.6 we have that $\Delta, \Delta', X \leq \delta, \Delta'' \vdash \tau\ \mathsf{ok}$. Thus, by definition of $\mathfrak{P}$, we know that $(\Delta, \Delta', X \leq \delta, \Delta''; \tau; \sigma) \in S(\mathfrak{P})$.

Case: $\sigma = \top$ **(Sub Top)**. Similar.

Case: $\tau = Y$ **(Sub TransVar)**

Independently of whether $X = Y$ or not we know that the tuple $(\Delta, X \leq \delta, \Delta', \Delta''; Y; \sigma) \in \nu S$ is supported by the tuple $(\Delta, X \leq \delta, \Delta', \Delta''; \tau'; \sigma) \in \nu S$ with $Y \leq \tau' \in \Delta, X \leq \delta, \Delta', \Delta''$ and therefore $Y \leq \tau' \in \Delta, \Delta', X \leq \delta, \Delta''$. By definition of S (Definition 5.8), $(\Delta, \Delta', X \leq \delta, \Delta''; Y; \sigma) \in S(\mathfrak{P})$.

Case: $\sigma = \mu Y.\sigma'$ **(Sub RecR)**

In this case, if $(\Delta, X \leq \delta, \Delta', \Delta''; \tau; \mu Y.\sigma') \in \nu S$ then $(\Delta, X \leq \delta, \Delta', \Delta''; \tau; \sigma'[Y \leftarrow \mu Y.\sigma']) \in \nu S$. By definition of $\mathfrak{P}$, $(\Delta, \Delta', X \leq \delta, \Delta''; \tau; \sigma'[Y \leftarrow \mu Y.\sigma']) \in \mathfrak{P}$ and by definition of S (Definition 5.8), $(\Delta, \Delta', X \leq \delta, \Delta''; \tau; \mu Y.\sigma') \in S(\mathfrak{P})$.

Case: $\tau = \mu Y.\tau'$ **(Sub RecL)**. Similar.

Case: $\tau = \tau = \forall_{X \leq \delta}\, \tau'$ **and** $\sigma = \forall_{X \leq \delta}\, \sigma'$ **(Sub All)**

This case is supported by the tuple $(\Delta, X \leq \delta, \Delta', \Delta'', X \leq \delta; \tau'; \sigma') \in \nu S$. By the definition of $\mathfrak{P}$, $(\Delta, \Delta', X \leq \delta, \Delta'', X \leq \delta; \tau'; \sigma') \in \mathfrak{P}$. And by definition of S (Definition 5.8), the tuple $(\Delta, \Delta', X \leq \delta, \Delta''; \tau; \sigma) \in S(\mathfrak{P})$.

Case: $\tau = \tau' \rightarrow \tau''$ **(Sub Fun)**. Similar.

We know that $\mathfrak{P} \subseteq \nu S$, hence if $\Delta, X \leq \delta, \Delta', \Delta'' \vdash \tau \leq \sigma$ and $X \notin FV(\Delta')$ then we have $\Delta, \Delta', X \leq \delta, \Delta'' \vdash \tau \leq \sigma$. □

(This is a repetition of Lemma 5.16, defined in page 121.)

Lemma 5.16 (νS is closed under transitivity)**.** *$\mathfrak{T} \subseteq \nu S$.*

Proof. The proof is done by the coinduction principle, showing that $\mathfrak{T}$ is *S-consistent* ($\mathfrak{T} \subseteq S(\mathfrak{T})$) and therefore $\mathfrak{T} \subseteq \nu S$. We will use an internal induction on the number of tuples of $\mathfrak{N}^n$ that form a chain from α_0 to α_n. In the proof, we write $\mathfrak{T}^i$ to denote the subset of $\mathfrak{T}$ containing only the tuples supported by chains of size i.

Case: $n = 1$

This is the induction base, for which we have $\mathfrak{T}^1 = \mathfrak{N}^1$ and therefore we also prove that νS is closed under narrowing. So, a tuple $(\Delta, \alpha_0, \alpha_1) \in \mathfrak{T}^1$ if $(\Gamma, \alpha_0, \alpha_1) \in \nu S$ and exists $\Gamma \in \mathcal{D}$ such that $\Delta \sqsubseteq_{\mathfrak{N}^0} \Gamma$, where by definition $\mathfrak{N}^0 = \nu S$. The proof is divided in the different cases that may support $(\Gamma, \alpha_0, \alpha_1) \in \nu S$.

Subcase: $\alpha_0 = \alpha_1$ **(Sub Equal) and** $\alpha_1 = \top$ **(Sub Top)**

In this case $\Gamma \vdash \alpha_0$ ok and we know by the definition of S that $(\Gamma, \alpha_0, \alpha_0) \in S(R)\ \forall R \in \mathcal{P}(\mathcal{J})$. Which means, in this case, that $(\Delta, \alpha_0, \alpha_1) \in S(\mathfrak{T})$ since $Dom(\Delta) = Dom(\Gamma)$, by Definition 5.12.

Subcase: $\alpha_0 = X$ **(Sub TransVar)**

$(\Delta, X, \alpha_1) \in \mathfrak{T}$ is supported by $(\Gamma, X, \alpha_1) \in \nu S$ with $\Delta \sqsubseteq_{\nu S} \Gamma$ and in turn by $(\Gamma, \Gamma(X), \alpha_1) \in \nu S$. By definition of $\mathfrak{N}^1$ (Definition 5.12) we also know that $(\Gamma, \Delta(X), \Gamma(X)) \in \nu S$ and therefore $(\Delta, \Delta(X), \Gamma(X))$, $(\Delta, \Gamma(X), \alpha_1) \in \mathfrak{N}^1$. Which means that, by definition of $\mathfrak{T}$, $(\Delta, \Delta(X), \alpha_1) \in \mathfrak{T}$ (in $\mathfrak{T}^2$). Finally, by the definition of S, we conclude that $(\Delta, X, \alpha_1) \in S(\mathfrak{T})$ by (Sub TransVar).

Subcase: $\sigma = \mu\alpha.\sigma'$ **(Sub RecR)**

In this case we know that if $(\Gamma; \tau; \mu X.\sigma') \in \nu S$ it is because $(\Gamma; \tau; \sigma'[X \leftarrow \mu X.\sigma']) \in \nu S$. By definition, $(\Delta; \tau; \sigma'[X \leftarrow \mu X.\sigma']) \in \mathfrak{N}^1$ and therefore in $\mathfrak{T}$ and $(\Delta; \tau; \mu X.\sigma') \in S(\mathfrak{T})$.

Subcase: $\tau = \mu X.\tau'$ **(Sub RecL)**. Similar..

Subcase: $\tau = \forall_{X \leq \delta}\ \tau'$ **(Sub All)**

In this case we have $\sigma = \forall_{X \leq \delta}\ \sigma'$ and that $(\Delta, \tau, \sigma) \in \mathfrak{T}$ is supported by the tuple $(\Gamma, \tau, \sigma) \in \nu S$ and more, $(\Gamma, X \leq \delta; \tau'; \sigma') \in \nu S$. By Definition 5.13, $(\Delta, , X \leq \delta; \tau'; \sigma') \in \mathfrak{N}^1$ and therefore in νS. Thus the conclusion $(\Delta; \tau; \sigma) \in S(\mathfrak{T})$ is true.

Subcase: $\tau = \tau' \to \tau''$ **(Sub Fun)**. Similar.

Case: $n > 1$

In the induction step we have a tuple $(\Delta, \alpha_0, \alpha_n) \in \mathfrak{T}$ which, by definition, is supported by a set of tuples $(\Delta, \alpha_{i-1}, \alpha_i) \in \mathfrak{N}^n$ ($i \in 1..n$). By definition of $\mathfrak{N}^n$ each one of these tuples is supported by a tuple $(\Gamma_i^n, \alpha_{i-1}, \alpha_i) \in \nu S$ such that $\Delta \sqsubseteq^n_{\mathfrak{N}^{n-1}..\mathfrak{N}^0} \Gamma_i^n$ with $i \in 1..n$ where $\Delta \sqsubseteq^n_{\mathfrak{N}^{n-1}..\mathfrak{N}^0} \Gamma_i^n = \Delta \sqsubseteq_{\mathfrak{N}^{n-1}} \Gamma_i^1 \sqsubseteq_{\mathfrak{N}^{n-2}} \ldots \sqsubseteq_{\mathfrak{N}^1} \Gamma_i^{n-1} \sqsubseteq_{\mathfrak{N}^0} \Gamma_i^n$. The proof is divided in the cases by which the first tuple $(\Gamma_1^n, \alpha_0, \alpha_1)$ may be supported in νS. The inner induction hypothesis is applied to cases based on smaller chains of tuples (with size $n-1$). We will refer to Γ_1^n as Γ^n when obvious from context.

Subcase: $\alpha_0 = \alpha_1$ **(Sub Equal)**

In this case there is a the set of $n-1$ tuples, $(\Delta, \alpha_0, \alpha_2), ..., (\Delta, \alpha_{n-1}, \alpha_n)$, which by induction hypothesis leads to $(\Delta, \alpha_0, \alpha_n) \in S(\mathfrak{T})$.

Case: $\alpha_0 = \top$ **(Sub Top)**. Similar.

Subcase: $\alpha_0 = X$ **(Sub TransVar)**

By the definitions of $\mathfrak{N}^n$ and νS we know that the tuple we are focusing on, $(\Delta, X, \alpha_1) \in \mathfrak{N}^n$, is supported by $(\Gamma^n, X, \alpha_1) \in \nu S$ which in turn is supported by $(\Gamma^n, \Gamma^n(X), \alpha_1) \in \nu S$. This implies by definition that $(\Delta, \Gamma^n(X), \alpha_1) \in \mathfrak{N}^n$. By definition of $\mathfrak{N}^n$ we also know that there is a sequence of narrowings:

$$(\Gamma^1, \Delta(X), \Gamma^1(X)) \in \mathfrak{N}^{n-1}, (\Gamma^2, \Gamma^1(X), \Gamma^2(X)) \in \mathfrak{N}^{n-2}, \cdots, (\Gamma^n, \Gamma^{n-1}(X), \Gamma^n(X)) \in \mathfrak{N}^0.$$

Thus, by definition of $\mathfrak{N}^n$ we conclude that there is a sequence of n tuples in $\mathfrak{N}^n$:

$$(\Delta, \Delta(X), \Gamma^1(X)), (\Delta, \Gamma^1(X), \Gamma^2(X)), \cdots, (\Delta, \Gamma^{n-1}(X), \Gamma^n(X)).$$

Another chain of size n in $\mathfrak{N}^n$ comprises the tuples:

$$(\Delta, \Gamma^n(X), \alpha_1), (\Delta, \alpha_1, \alpha_2), \cdots, (\Delta, \alpha_{n-1}, \alpha_n).$$

By definition of $\mathfrak{T}$ we conclude that $(\Delta, \Delta(X), \alpha_n) \in \mathfrak{T}$ (in $\mathfrak{T}^{2n}$) and therefore $(\Delta, X, \alpha_n) \in S(\mathfrak{T})$.

Subcase: $\alpha_1 = \mu X.\beta_1$ **(Sub RecR)**

In this case we know that if $(\Gamma_1^n; \alpha_0; \mu X.\beta_1) \in \nu S$ it is because $(\Gamma_1^n; \alpha_0; \beta_1[X \leftarrow \alpha_1]) \in \nu S$. As for the second tuple $(\Gamma_2^n; \alpha_1; \alpha_2) \in \nu S$, it can be supported either by the left recursion or by right recursion case of S. In the first case we have that $(\Gamma_2^n; \beta_1[X \leftarrow \alpha_1]; \alpha_2) \in \nu S$, and in the second $\alpha_2 = \mu X.\beta_2$ and $(\Gamma_2^n; \alpha_1; \beta_2[X \leftarrow \alpha_2]) \in \nu S$ which in turn can only be supported by $(\Gamma_2^n; \beta_1[X \leftarrow \alpha_1]; \beta_2[X \leftarrow \alpha_2]) \in \nu S$. Continuing to the next tuple we can replace each tuple by its unfolding until we rebuild the chain from α_0 to α_n or the corresponding unfolding.

If the last tuple of the chain is $(\Gamma_n^n; \alpha_{n-1}; \beta_n[X \leftarrow \alpha_n])$ then we have a chain in $\mathfrak{N}^n$ that sustains that $(\Delta; \alpha_0; \beta_n[X \leftarrow \alpha_n]) \in \mathfrak{T}$ and therefore $(\Delta; \alpha_0; \alpha_n) \in S(\mathfrak{T})$. If this is not the case then the support for the first tuple, $(\Gamma_1^n; \alpha_0; \beta_1[X \leftarrow \alpha_1]) \in \nu S$ leads to the result that $(\Delta; \alpha_0; \alpha_n) \in S(\mathfrak{T})$ as we can see in all other cases in this proof (this particular case as α_1 is contractive and may not be included in that support).

Subcase: $\alpha_0 = \mu X.\beta_0$ **(Sub RecR)**

In this case we have that the support for the first tuple is $(\Gamma_1^n; \beta_0[X \leftarrow \alpha_0]; \alpha_1) \in \nu S$ and therefore $(\Delta; \beta_0[X \leftarrow \alpha_0]; \alpha_n) \in \mathfrak{T}$. Then we have that $(\Delta; \alpha_0; \alpha_n) \in S(\mathfrak{T})$.

Subcase: $\alpha_0 = \forall_{X \leq \delta}\, \alpha_0'$ **and** $\alpha_1 = \forall_{X \leq \delta}\, \alpha_1'$ **(Sub All)**

The tuple supporting $(\Gamma, \alpha_0, \alpha_1)$ is $(\Gamma_1^n, X \leq \delta; \alpha_0'; \alpha_1') \in \nu S$ with $\Delta \sqsubseteq^n_{\mathfrak{N}_{n-1}..\mathfrak{N}^0} \Gamma_1^n$. By inspection of the generating function we see that the shapes of all α_i must be the same (quantifier constructor) and the possibilities for supporting the tuples are either reflexivity (Sub Equal) or the class type constructor (Sub All). If it is the case that one of the tuples is supported by reflexivity then $\exists j \in 1..n-1$ such that $\alpha_j = \alpha_{j+1}$ and there is a chain of tuples with size $n-1$ from α_0 to α_n. By induction hypothesis we know that $(\Delta, \alpha_0, \alpha_n) \in S(\mathfrak{T})$.

If one the other hand all tuples are supported by the (Sub All) case, let all $\alpha_i = \forall_{X \leq \delta}\, \alpha_i'$ In this case, $(\Delta, \alpha_i, \alpha_{i+1}) \in \mathfrak{N}^n$ for all $i \in 1..n-1$ are supported in νS by $(\Gamma_i^n, \alpha_i, \alpha_{i+1}) \in \nu S$ for all $i \in 1..n-1$ and their support comprises the tuples $(\Gamma_i^n, X \leq \delta; \alpha_i'; \alpha_{i+1}') \in \nu S$ with $\Delta \sqsubseteq^n_{\mathfrak{N}_{n-1}..\mathfrak{N}^0} \Gamma_1^n$. By the definition of $\mathfrak{N}^n$ we conclude that $(\Delta, X \leq \delta; \alpha_i'; \alpha_{i+1}') \in \mathfrak{N}^n$ with $\Delta \sqsubseteq^n_{\mathfrak{N}_{n-1}..\mathfrak{N}^0} \Gamma_1^n$. And by definition of $\mathfrak{T}$ $(\Delta, X \leq \delta; \alpha_0'; \alpha_n') \in \mathfrak{T}$ and therefore $(\Delta, \alpha_0, \alpha_n) \in S(\mathfrak{T})$.

Subcase: $\tau = \tau' \rightarrow \tau''$ **(Sub Fun)**. Similar. □

(This is a repetition of Lemma 5.18, defined in page 122.)

Lemma 5.18 (Substitution of type variables)**.** *For all $\Delta, \Delta' \in \mathcal{D}$, and $\tau, \sigma, \delta, \delta' \in \mathcal{T}_\lambda$, if $\Delta, X \leq \delta, \Delta' \vdash \tau \leq \sigma$ and $\Delta \vdash \delta' \leq \delta$ then we have $\Delta, \Delta'[X \leftarrow \delta'] \vdash \tau[X \leftarrow \delta'] \leq \sigma[X \leftarrow \delta']$.*

Proof. We first define an auxiliary set that represents the closure of νS under substitution.

$$\mathfrak{Q} \triangleq \nu S \cup \{(\Delta, \Delta'[X \leftarrow \delta']; \tau[X \leftarrow \delta']; \tau'[X \leftarrow \delta']) \mid (\Delta, X \leq \delta', \Delta'; \tau; \sigma), (\Delta, \delta', \delta) \in \nu S\}$$

And then prove that $\mathfrak{Q} \subseteq \nu S$. By coinduction principle it is sufficient to prove that $\mathfrak{Q}$ is S-*consistent*. We now divide the proof in the cases of $\mathfrak{Q}$. If an element is in $\mathfrak{Q}$ by the first case then it is obvious that it is also a member of νS. On the second case, for each tuple $(\Delta, \Delta'[X \leftarrow \delta']; \tau[X \leftarrow \delta']; \sigma[X \leftarrow \delta']) \in \mathfrak{Q}$ there is a pair of tuples $(\Delta, X \leq \delta, \Delta'; \tau; \sigma), (\Delta; \delta'; \delta) \in \nu S$ that support it. In order to prove that $(\Delta, \Delta'[X \leftarrow \delta']; \tau[X \leftarrow \delta']; \sigma[X \leftarrow \delta']) \in S(\mathfrak{Q})$ we divide the proof in the cases that support $(\Delta, X \leq \delta', \Delta'; \tau; \sigma) \in \nu S$.

Case: $\tau \equiv \sigma$ **(Sub Equal)**

In this case we know that $\Delta, X \leq \delta', \Delta' \vdash \tau$ ok By Lemma 5.5, we have that $\Delta, \Delta'[X \leftarrow \delta'] \vdash \tau[X \leftarrow \delta']$ ok. By definition of S, $(\Delta, \Delta'[X \leftarrow \delta']; \tau[X \leftarrow \delta']; \tau[X \leftarrow \delta']) \in S(\mathfrak{Q})$.

Case: $\tau \equiv Y$ **(Sub TransVar)**

The support for the tuple $(\Delta, \Delta'[X \leftarrow \delta']; Y; \sigma[X \leftarrow \delta']) \in \mathfrak{Q}$ is a tuple $(\Delta, X \leq \delta, \Delta'; Y; \sigma) \in \nu S$ which is in turn supported by the tuple $(\Delta, X \leq \delta, \Delta'; \tau'; \sigma) \in \nu S$ with $Y \leq \tau' \in \Delta, X \leq \delta, \Delta'$. This last tuple supports $(\Delta, \Delta'[X \leftarrow \delta']; \tau'[X \leftarrow \delta']; \sigma[X \leftarrow \delta']) \in \mathfrak{Q}$. We then have to analyse three different subcases:

Subcase: $Y \leq I \in \Delta$

we have that X does not occur in τ' and $\tau'[X \leftarrow \tau'] = \tau'$ and $(\Delta, \Delta[X \leftarrow \delta']; Y; \sigma[X \leftarrow \delta']) \in S(\mathfrak{Q})$.

Subcase: $Y \leq \tau' \in \Delta'$

We know that $Y \leq \tau'[X \leftarrow \delta'] \in \Delta'[X \leftarrow \delta']$ and therefore $(\Delta, \Delta[X \leftarrow \delta']; Y; \sigma[X \leftarrow \delta']) \in S(\mathfrak{Q})$.

Subcase: $X \equiv Y$

By the definition of $\mathfrak{Q}$ we know $(\Delta, X \leq \delta, \Delta'; X; \sigma), (\Delta, \delta', \delta) \in \nu S$, and in the case of support by variable transitivity in X we also know that the tuple $(\Delta, X \leq \delta, \Delta'; \delta; \sigma) \in \nu S$. By Lemma 5.11 (weakening) we also know that $(\Delta, X \leq \delta, \Delta'; \delta'; \delta) \in \nu S$. By Lemma 5.16 (transitivity) we know that the tuple $(\Delta, X \leq \delta, \Delta'; \delta'; \sigma) \in \nu S$. Lets now analyse the possible cases of S that may support this new tuple.

(Sub Equal) From $\delta' \equiv \sigma$ we know that $\Delta \vdash \delta'$ ok and by Lemma 5.4 we know that $\Delta, \Delta'[X \leftarrow \delta'] \vdash \delta'$ ok. As $\delta'[X \leftarrow \delta'] = \delta'$ we have that, by definition of S, $(\Delta, \Delta'[X \leftarrow \delta']; \delta'[X \leftarrow \delta']; \delta'[X \leftarrow \delta']) \in S(\mathfrak{Q})$ which is exactly the conclusion we were looking for when $\delta' \equiv \sigma$ and $\tau \equiv X$.

(Sub TransVar) In this case we have that $\delta' \equiv Z$ and $Z \not\equiv X$. We also know that $Z \leq \delta'' \in \Delta$ and $(\Delta, X \leq \delta, \Delta'; \delta''; \sigma) \in \nu S$. By definition $(\Delta, \Delta'[X \leftarrow \delta']; \delta''[X \leftarrow \delta']; \sigma[X \leftarrow \delta']) \in \mathfrak{Q}$ with $\delta''[X \leftarrow \delta'] \equiv \delta''$. Therefore $(\Delta, \Delta'[X \leftarrow \delta']; Z; \sigma[X \leftarrow \delta']) \in S(\mathfrak{Q})$ which is the result we were looking for. Note that $\tau[X \leftarrow \delta'] \equiv Z$.

(Sub All) In this case $\delta' \equiv \forall_{Y \leq \gamma} \delta''$ and also $\sigma \equiv \forall_{Y \leq \gamma} \sigma'$. By the definition of S we know that the tuple $(\Delta, X \leq \delta, \Delta', Y \leq \gamma; \delta''; \sigma') \in \nu S$. Then, by the definition of $\mathfrak{Q}$ we know that $(\Delta, \Delta'[X \leftarrow \delta'], Y \leq \gamma; \delta''[X \leftarrow \delta'], \sigma'[X \leftarrow \delta']) \in \mathfrak{Q}$. As X does not occur in δ', $(\Delta, \Delta'[X \leftarrow \delta']; \delta'[X \leftarrow \delta']; \sigma[X \leftarrow \delta']) \in S(\mathfrak{Q})$. Remember that X does not occur in δ' and the intended conclusion is $(\Delta, \Delta'[X \leftarrow \delta']; X[X \leftarrow \delta']; \sigma[X \leftarrow \delta']) \in S(\mathfrak{Q})$.

(Sub Fun) Similar.

All this subcases prove the lemma for the case where the tuple is supported by variable transitivity.

Case: $\tau \equiv \forall_{Y \leq \gamma} \tau'$ **and** $\sigma \equiv \forall_{Y \leq \gamma'} \sigma'$ **(Sub All)**

To prove that $(\Delta, \Delta'[X \leftarrow \delta']; \tau[X \leftarrow \delta']; \sigma[X \leftarrow \delta']) \in S(\mathfrak{Q})$ we start with the tuples $(\Delta, X \leq \delta, \Delta'; \tau; \sigma), (\Delta, \delta', \delta) \in \nu S$ where the first tuple is supported by $(\Delta, X \leq \delta, \Delta', Y_j \leq \gamma; \tau', \sigma') \in \nu S$. By definition we can say that $(\Delta, \Delta'[X \leftarrow \delta'], Y \leq \gamma; \tau'[X \leftarrow \delta'], \sigma'[X \leftarrow \delta']) \in \mathfrak{Q}$. Then $(\Delta, \Delta'[X \leftarrow \delta']; \tau[X \leftarrow \delta']; \sigma[X \leftarrow \delta']) \in S(\mathfrak{Q})$.

Case: $\tau \equiv \tau' \rightarrow \tau''$ **and** $\sigma \equiv \sigma' \rightarrow \sigma''$ **(Sub Fun)**. Similar.

Finally if $\Delta, X \leq \delta, \Delta' \vdash \tau \leq \sigma$ and $\Delta \vdash \delta' \leq \delta$ then we have $(\Delta, X \leq \delta, \Delta'; \tau; \sigma)$, $(\Delta; \delta'; \delta) \in \nu S$. As $\mathfrak{Q} \subseteq \nu S$ we know that $(\Delta, \Delta'[X \leftarrow \delta']; \tau[X \leftarrow \delta']; \sigma[X \leftarrow \delta']) \in \nu S$ which is the same as $\Delta, \Delta'[X \leftarrow \delta'] \vdash \tau[X \leftarrow \delta'] \leq \sigma[X \leftarrow \delta']$. □

(This is a repetition of Lemma 5.19, defined in page 122.)

Lemma 5.19 (Equivariance)**.** *For all* $\Delta \in \mathcal{P}(\mathcal{J}), \tau, \sigma \in \mathcal{T}_\lambda$*, if* $\Delta \vdash \tau \leq \sigma$ *then* $\Delta[X \leftrightarrow Y] \vdash \tau[X \leftrightarrow Y] \leq \sigma[X \leftrightarrow Y]$.

Proof. From $\Delta \vdash \tau \leq \sigma$ and assuming that $X, Y \in Dom(\Delta)$ we have that $\Delta = \Delta', X \leq \delta, \Delta'', Y \leq \gamma, \Delta'''$. We prove this lemma by dividing the varible replacing $[X \leftrightarrow Y]$ in a triplet of substitutions $[X \leftarrow Z][Y \leftarrow X][Z \leftarrow Y]$ where Z is fresh. By weakening (Lemma 3.24), we have

$$\Delta', Z \leq \delta, X \leq \delta, \Delta'', Y \leq \gamma, \Delta''' \vdash \tau \leq \sigma.$$

By the substitution lemma we have that

$$\Delta', Z \leq \delta, \Delta''[X \leftarrow Z], Y \leq \gamma[X \leftarrow Z], \Delta'''[X \leftarrow Z] \vdash \tau[X \leftarrow Z] \leq \sigma[X \leftarrow Z].$$

Again, by weakening we obtain

$$\Delta', Z \leq \delta, \Delta''[X \leftarrow Z], X' \leq \gamma[X \leftarrow Z], Y \leq \gamma[X \leftarrow Z], \Delta'''[X \leftarrow Z] \vdash \tau[X \leftarrow Z] \leq \sigma[X \leftarrow Z].$$

Notice that we have use X' to represent the new X without loss of generacity because we keep them iin different scopes. By the substitution lemma, we have

$$\begin{aligned}&\Delta', Z \leq \delta, \Delta''[X \leftarrow Z], X' \leq \gamma[X \leftarrow Z], \Delta'''[X \leftarrow Z][Y \leftarrow X'] \vdash \\ &\qquad\qquad \tau[X \leftarrow Z][Y \leftarrow X'] \leq \sigma[X \leftarrow Z][Y \leftarrow X'].\end{aligned}$$

Using the same weakening/substitution scheme we obtain

$$\begin{aligned}&\Delta', Y' \leq \delta, \Delta''[X \leftarrow Z][Z \leftarrow Y'], X' \leq \gamma[X \leftarrow Z][Z \leftarrow Y'], \Delta'''[X \leftarrow Z][Y \leftarrow X'][Z \leftarrow Y'] \vdash \\ &\qquad\qquad \tau[X \leftarrow Z][Y \leftarrow X'][Z \leftarrow Y'] \leq \sigma[X \leftarrow Z][Y \leftarrow X'][Z \leftarrow Y'].\end{aligned}$$

which is the same as

$$\Delta[X \leftrightarrow Y] \vdash \tau[X \leftrightarrow Y] \leq \sigma[X \leftrightarrow Y].$$

□

A.4 Chapter 5

(This is a repetition of Lemma 6.22, defined in page 154.)

Lemma 6.22 (Subject Reduction)**.**

1. *Let $(e; S)$ be a valid configuration in $\lambda_{\chi}^{\leq} \backslash \{\mathsf{nil}\}$ such that $\mathsf{nil}(S) = \emptyset$ and let Γ be a typing environment typing S such that e and S are record-based with relation to Γ.*

   If $\Gamma \vdash e : \tau$ and $(e; S \downarrow v; S')$ then

   a) there is a Γ' that extends Γ and types S',

   b) $\Gamma' \vdash v : \tau$ such that $\Delta \vdash \tau' \leq \tau$,

   c) v is either an abstraction, a component, a configurator, or a location that is either `undefined` or `refers-to` a record,

   d) v and S' are record-based with relation to Γ, and

   e) $\mathsf{nil}(S') = \emptyset$.

2. *Let c be an expression such that $\Gamma \vdash c : K \Longrightarrow K'$, let S be a heap such that, for some set $X \in Dom(S)$, $\mathsf{nil}(S) \subseteq \{\text{select}_S(s,\pi) \mid (\pi:\tau) \in K_\triangleleft \cup K_\circ\} \uplus X$ and Γ types S.*

   Let s be a partially linked object s such that it complies with K and its partially linked object type is $[\![R \oplus K_\triangleleft \Rightarrow P \oplus K_\triangleright]\!]$ and s and S are record-based with relation to Γ:

   If $s; c; S \Downarrow s'; S'$ then

   a) *there is Γ' typing S' and extending Γ,*

   b) *s' is a partially linked object that extends s and complies with K'. Its partially linked object type is $[\![R \oplus K'_\triangleleft \Rightarrow P \oplus K'_\triangleright]\!]$, and*

   c) *s' and S' are record-based with relation to Γ', and*

   d) *$\mathsf{nil}(S') \subseteq \{\text{select}_{S'}(s,\pi) \mid (\pi:\tau) \in K'_\triangleleft \cup K'_\circ\} \uplus X$.*

Proof. The proof is carried out by induction on the two cases of the lemma. On the first case, we prove it by induction on the size of the evaluation derivations and by analysis of the last rule used. We use the second case when needed and its conditions are met. We show, in the cases that evaluate to wrong, that these rules are never applicable.

***Case:* (Eval Value)**

The conclusions of the theorem hold with $\Gamma' = \Gamma$, $S' = S$, and $v = \lambda x : \delta.e$ or $v = \Lambda X \leq \delta.e$. $\Gamma \vdash v : \tau$ with $\Gamma \vdash \tau \leq \tau$ by definition of subtyping, Definition 5.10. We also have that $\mathsf{nil}(S') = \emptyset$ and that both v and S are record-based.

***Case:* (Eval Application)$^\tau$**

In this case, with $e_1(e_2); S \downarrow v; S'$, we know by Rule (Eval Application), that: a) $e_1; S \downarrow \lambda x : \tau.e; S'$, b) $e_2; S' \downarrow v_2; S''$, and c) $e[x \leftarrow v_2]; S'' \downarrow v; S'''$. On the typing derivation, the only plausible last rule for $\Gamma \vdash e_1(e_2) : \sigma$ is Rule , with the premises: d) $\Gamma \vdash e_1 : \tau \rightarrow \sigma$ and e) $\Gamma \vdash e_2 : \tau$.

Taking d) and a), by induction hypothesis we conclude that there is a Γ' extending Γ and typing S', such that f) $\Gamma' \vdash \lambda x : \tau.e : \tau' \rightarrow \sigma'$ with $\Gamma' \vdash \tau' \rightarrow \sigma' \leq \tau \rightarrow \sigma$. By Lemma 3.24 (weakening) on e) we conclude that $\Gamma' \vdash e_2 : \tau$, which together with b), by induction hypothesis, there is a Γ'' extending both Γ' and Γ and typing S'' such that g) $\Gamma'' \vdash v_2 : \tau''$ with $\Gamma'' \vdash \tau'' \leq \tau$ and $\mathsf{nil}(S'') = \emptyset$. From f), by Lemma 3.24 (weakening) and Rule (Val Abstraction) we have that $\Gamma'', x : \tau \vdash e : \sigma$ and by Lemma 3.28 (substitution) with g) we have that h) $\Gamma'' \vdash e[x \leftarrow v_2] : \sigma'$ with $\Gamma'' \vdash \sigma' \leq \sigma$. Taking h) and c), by induction hypothesis we have that there is Γ''' extending Γ'' and typing S''', such that v is typed $\Gamma''' \vdash v : \sigma$ and v is either an abstraction or a location that is either *`undefined`* or *`refers-to`* a record, and $\mathsf{nil}(S''') = \emptyset$.

The application of induction hypothesis to the different premises and the application of Lemma 3.28 ensures that v and S''' are record-based.

***Case:* (Eval Record)**

When the last rule is (Eval Record) we have the hypothesis $[\ell_i = e_i\ ^{i\in 1..n}]; S_0 \downarrow \imath; S'$ with $S' = S_n[\imath_i \mapsto v_i\ ^{i\in 1..n}][\imath \mapsto \{\ell_i = \imath_i\ ^{i\in 1..n}\}]$ and $\Gamma \vdash [\ell_i = e_i\ ^{i\in 1..n}] : \{\!|\ell_i : \tau_i\ ^{i\in 1..n}|\!\}$. By the Rules (Val Record) and (Eval Record), we have that a) $\Gamma \vdash e_i : \tau_i$ and b) $e_i; S_{i-1} \downarrow v_i; S_i$ with $i \in 1..n$. By iterating the induction hypothesis with a) and b), and using Lemma 3.24 (weakening) to adjust the typing environment in the hypothesis, we reach the conclusion that, for all $i \in 1..n$, there is a Γ_i extending Γ_{i-1} and typing each S_i, with $\Gamma_0 = \Gamma$ such that $\Gamma_i \vdash v_i : \tau_i'$ with $\Gamma_i \vdash \tau_i' \leq \tau_i$ and $\mathsf{nil}(S_i) = \emptyset$. So, by transitivity, we have that Γ_n types S_n and extends Γ with $\mathsf{nil}(S_n) = \emptyset$. By Rule (Val Record Value) and Definition 2.21 we have that $\Gamma' = \Gamma_n, \imath : \{\!|\ell_i : \tau_i\ ^{i\in 1..n}|\!\}$ types S' and that $\Gamma' \vdash \imath : \{\!|\ell_i : \tau_i\ ^{i\in 1..n}|\!\}$ with relation to S'. We have that $\imath$ is a location that `refers-to` a record and $\mathsf{nil}(S') = \emptyset$. The resulting heap is also record-based since all the introduced values are record-based (by induction hypothesis).

***Case:* (Eval Assign)**

In this case, from $e_1.\ell := e_2$, by Rule (Eval Assign), we have that a) $e_1; S \downarrow \imath; S'$ with $\imath' = \mathrm{deref}_{S'}(\imath)$ and $S'(\imath') = \{\ldots, \ell = \imath'', \ldots\}$, and b) $e_2; S' \downarrow v; S''$. By Rule (Val Assign), we have that $\Gamma \vdash (e_1.\ell := e_2) : \sigma$ is supported by c) $\Gamma \vdash e_1 : \{\!|\ldots, \ell : \sigma, \ldots|\!\}$ and d) $\Gamma \vdash e_2 : \sigma$. Thus, from a) and c), by induction hypothesis we know that there is a Γ' extending Γ and typing S' such that $\Gamma' \vdash \imath : \{\!|\ldots, \ell : \sigma, \ldots|\!\}$, and $\mathsf{nil}(S') = \emptyset$. Notice that record types are only related by equivalence. By Definition 2.21 we conclude that $S'(\mathrm{deref}_{S'}(\imath)) = \{\ldots, \ell = \imath'', \ldots\}$ and that $\Gamma \vdash S'(\mathrm{deref}_{S'}(\imath)) : \{\!|\ldots, \ell : \sigma, \ldots|\!\}$. By rule (Val Record) we conclude that e) $\Gamma' \vdash \imath'' : \sigma$. From b), by Lemma 3.24 (weakening) we obtain $\Gamma' \vdash e_2 : \sigma$, and by induction hypothesis together with d) we have that there is a Γ'' extending Γ' and typing S'' such that f) $\Gamma'' \vdash v : \sigma'$ with $\Gamma'' \vdash \sigma' \leq \sigma$, and $\mathsf{nil}(S'') = \emptyset$. Γ' types $S''[\imath \mapsto v]$ and the result of the assignment expression is v, and v is not nil we conclude this case with $\mathsf{nil}(S''[\imath \mapsto v]) = \emptyset$ and v satisfying the conclusions of the theorem. Value v is record-based and it is introduced inside a record, so the resulting heap $S''[\imath \mapsto v]$ is also record-based.

***Case:* (Eval Select)**

If the last rule used is (Eval Select) then the expression $e.\ell$ can be typed by two possible rules. In both cases we have that a) $e_1; S \downarrow \imath; S'$ with $\imath' = \mathrm{deref}_{S'}(\imath)$ and $S'(\imath') = \{\ldots, \ell = \imath'', \ldots\}$. The subcases are:

***Subcase:* (Val Select)**

Now, in this case we have that c) $\Gamma \vdash e : \{\!|\ldots, \ell : \tau, \ldots|\!\}$. From a) and c), by induction hypothesis we know that there is a Γ' extending Γ and typing S' such that $\Gamma' \vdash \imath : \{\!|\ldots, \ell : \sigma, \ldots|\!\}$ and $\mathsf{nil}(S') = \emptyset$. Notice again that record types are only related by equivalence. By Definition 2.21 we conclude that for some $\imath' = \mathrm{deref}_{S'}(\imath)$ we have $S'(\imath') = \{\ldots, \ell = \imath'', \ldots\}$ and that $\Gamma \vdash S'(\imath') : \{\!|\ldots, \ell : \sigma, \ldots|\!\}$. By Rule (Val Record) we conclude that e) $\Gamma' \vdash \imath'' : \sigma$. By Definition 2.21

we conclude that $\Gamma' \vdash S'(\imath'') : \sigma$. Notice that $\mathsf{nil}(S') = \emptyset$. The resulting value results from consulting a record in a record-based heap, so, it is also record-based.

***Subcase:* (Val Select Interface)**

In this case we know that c) $\Gamma \vdash e : \{\ldots, \ell : \tau, \ldots\}$ and therefore, by induction hypothesis (from a) and c)), that e_1 yields an interface typed value $(\imath)$, $\Gamma \vdash \tau' \leq \{\ldots, \ell : \tau, \ldots\}$. By inspection of the type system (Definition 3.18) we see, by Rule (Val Select Interface), that $\Gamma \vdash \imath : \{\!|\ldots, \ell : \tau, \ldots|\!\}$ and then the reasoning follows by applying Definition 2.21 and Rule (Val Record) as in the subcase above.

***Case:* (Eval Compose)**

We have the typing $\Gamma \vdash \mathsf{compose}\ e : K_{\triangleleft} \Rightarrow K_{\triangleright}$ supported by $\Gamma \vdash e : \emptyset \Longrightarrow K$ where $K_{\circ} = \emptyset$. The evaluation hypothesis is $\mathsf{compose}\ e; S \downarrow \mathsf{comp}(c); S'$ where $e; S \downarrow \mathsf{conf}(\tau, c); S'$. By induction hypothesis we have that there is a Γ' extending Γ, that types S' and $\Gamma' \vdash \mathsf{conf}(\tau, c) : \emptyset \Longrightarrow K$ and $\mathsf{nil}(S') = \emptyset$. Notice that configurator types are only related by equivalence. By Rule (Val Configurator Value)$^{\rho}$ we know that $\Gamma' \vdash c : \tau$ and that $\tau = \emptyset \Longrightarrow K$, and by Rules (Val Compose) and (Val Composition Value) we conclude that $\Gamma' \vdash \mathsf{comp}(c) : K_{\triangleleft} \Rightarrow K_{\triangleright}$. with Γ' extending Γ and typing S', and with the resulting value being a component value. From the application of the induction hypothesis we also conclude that $\mathsf{nil}(S') = \emptyset$ and that the value $comp(c)$ is record-based.

***Case:* (Eval New)**

The hypothesis $\Gamma \vdash \mathsf{new}\ e\ \mathsf{with}\ \ell^r_j := e_j\ ^{j \in 1..n} : \{\ell^p_i : \sigma_i\ ^{i \in 1..n}\}$ is obtained by Rule (Val New) with the premises $\Gamma \vdash e : \tau$ with $\tau = \{\ell^r_j : \tau_j\ ^{j \in 1..m}\} \Rightarrow \{\ell^p_i : \sigma_i\ ^{i \in 1..n}\}$. new is evaluated by Rule (Eval New), thus, if $\mathsf{new}\ e\ \mathsf{with}\ \ell^r_j := e_j\ ^{j \in 1..m}; S \downarrow \imath; S'$, then we must have $e; S \downarrow \mathsf{comp}(c); S_0$. By induction hypothesis we have that there is a Γ_0 extending Γ such that $\Gamma_0 \vdash \mathsf{comp}(c) : \tau'$ with $\Gamma_0 \vdash \tau' \leq \tau$ and $\mathsf{nil}(S_0) = \emptyset$.

Now, notice that the presentation order of the premises does not reflect the real dependence between them. For technical reasons, the plug-assignments must be considered first in this proof. By iteratively applying Lemma 4.11 (substitution), the induction hypothesis, and Lemma 4.10 (weakening) on the typing and evaluation judgements of each plug-assignment expression, $(\Gamma \vdash e_i : \tau_i)$ and $(e_i; S_{i-1} \downarrow v_i; S_i)$, we obtain that for all $i \in 1..n$ we have Γ_i extending Γ_{i-1} and typing S_i such that $\Gamma_i \vdash v_i : \tau'_i$ with $\Gamma_i \vdash \tau'_i \leq \tau_i$ and $\mathsf{nil}(S_i) = \emptyset$. Notice that all v_i and S_i are record-based.

By Lemma 4.10 (weakening) on the typing of $\mathsf{comp}(c)$ and Rule (Val Composition Value) we have $\Gamma_n \vdash c : \emptyset \Longrightarrow K$ and $K_{\circ} = \emptyset$, with type $\tau = K_{\triangleleft} \Rightarrow K_{\triangleright}$ thus $K_{\triangleleft} = \{\ell_j : \tau_j\ ^{j \in 1..m}\}$. Notice that the judgement above applies c to the empty instance $(\mathbf{0}; c; S_n \Downarrow s; S_{n+1})$, and that $\mathsf{nil}(S_n) = \emptyset$ meets the conditions of the second part of the lemma with $X = \emptyset$. In this case, $\mathbf{0}$ complies with the resource list $\emptyset$ and has type $[\![\{\} \Rightarrow \{\}]\!]$. By induction hypothesis on the second case of the lemma, we have that there is a Γ_{n+1} extending Γ_n and the resulting instance s is compliant with K with the partial type $[\![K_{\triangleleft} \Rightarrow K_{\triangleright}]\!]$. We know that the locations of the required ports are $\{\, \iota_i\ ^{i\in 1..n}\} = \{\mathsf{select}_{S'}(s,\ell) \mid (\ell:\tau) \in K_{\triangleleft}\}$ and that $\mathsf{nil}(S') \subseteq \{\mathsf{select}_{S'}(s,\ell) \mid (\ell:\tau) \in K_{\triangleleft}\}$.

Γ_{n+1} types S_{n+1} and the locations in $s_{\triangleleft} = \{\ell_i = \iota_i\ ^{i\in 1..n}\}$ are in S_n. From the compliance of s with K, Definition 3.23, and Rule (Val Record) we know that $\Gamma_{n+1} \vdash s : \{\!|\, \ldots, \ell_j^p : \sigma_j\ ^{j\in 1..m} |\!\}$ where $K_{\triangleright} = \{\ell_j^p : \sigma_j\ ^{j\in 1..m}\}$ and by Rule (Val Interface), we have a $\Gamma' = \Gamma_{n+1}, \iota : K_{\triangleright}$, extending Γ, and typing $S_{n+1}[\,\iota \mapsto s][\,\iota_i{\mapsto}v_i\ ^{i\in 1..n}]$. We finally have that ι is a location that `refers-to` an object, which is a record, such that $\Gamma' \vdash \iota : K_{\triangleright}$ with $K_{\triangleright} = \{\ell_j^p : \sigma_j\ ^{j\in 1..m}\}$. Finally, since $v_i \neq \mathsf{nil}$ for all $i \in 1..n$ we have that $\mathsf{nil}(S_{n+1}[\,\iota \mapsto s][\,\iota_j{\mapsto}v_j\ ^{j\in 1..m}]) = \emptyset$ with s being the newly created instance. Notice that ι and $S_{n+1}[\,\iota \mapsto s][\,\iota_j{\mapsto}v_j\ ^{j\in 1..m}]$ are record-based.

***Case:* (Eval Reconfig)**

If the last rule used is (Eval Reconfig) then we must have

a) $\mathsf{reconfig}\ x = e_1[e_2]\ \mathsf{with}\ \ell_i := f_i\ ^{i\in 1..n}\ \ \mathsf{in}\ e_3\ \mathsf{else}\ e_4; S \downarrow v; S'''$

and b) $\Gamma \vdash \mathsf{reconfig}\ x = e_1[e_2]\ \mathsf{with}\ \ell_i := e_i'\ ^{i\in 1..n}\ \mathsf{in}\ e_3\ \mathsf{else}\ e_4 : \delta$.

From b), by Rule (Val Reconfig), we have that $\Gamma \vdash e_1 : K \Longrightarrow K'$ and from a), by Rule (Eval Reconfig), we have $e_1; S \downarrow \mathsf{conf}(\tau', c); S'$. By induction hypothesis, we conclude that there is a Γ' that extends Γ and types S' such that $\Gamma' \vdash \mathsf{conf}(\tau', c) : K \Longrightarrow K'$, which, by Rule (Val Configurator Value), leads to $\tau' = K \Longrightarrow K'$ and $\Gamma' \vdash c : \tau'$. Notice that configurator types are only related by equivalence. From b), by Rule (Val Reconfig), we also have that $\Gamma \vdash e_2 : \tau$. By Lemma 4.10 (weakening) we obtain $\Gamma' \vdash e_2 : \tau$, and $e_2; S' \downarrow \iota; S_0$ with $S_0(\,\iota) = s$. By induction hypothesis we conclude that there is a Γ'' which extends Γ' and types S_0 such that $\Gamma'' \vdash \iota : \tau'$ with $\Gamma'' \vdash \tau' \leq \tau$.

Again, from a), by Rule (Eval Reconfig), we now know that $s \,/\!/\, K$ and therefore, by Lemma 4.9, we know that s complies with K and should have the partial linked object type $[\![\emptyset \Rightarrow \tau]\!]$.

The evaluation follows on the plug assigment expressions. By iterating Lemma 4.10 (weakening) and the induction hypothesis for all the expressions e_i, typed by $\Gamma_i \vdash e_i : \sigma_i$, we obtain that for each i there is a Γ_i that extends Γ_{i-1}, with $\Gamma_0 = \Gamma''$ typing each S_i such that $\Gamma_i \vdash v_i : \sigma_i'$ with $\Gamma_i \vdash \sigma_i' \leq \sigma_i$.

Then, we have that $s; c; S_n \Downarrow s'; S''$. By induction hypothesis on the second part of the lemma (the application of a configurator) we have that there is a Γ''' which extends Γ'' and that the resulting instance, s', complies with K' and has the partial type $[\![K'_{\triangleleft} \Rightarrow K'_{\triangleright} \oplus \tau]\!]$ and by Definition 3.21, its instance type is $K'_{\triangleright} \oplus \tau$.

Let $\Gamma'''' = \Gamma''', \iota' : (\tau \oplus K'_{\triangleright})$. Again, from b), by Rule (Val Reconfig) and Lemma 4.10 (weakening), we obtain $\Gamma'''', x : (\tau \oplus K'_{\triangleright}) \vdash e_3 : \delta$ and by Lemma 4.11 (substitution) with the side condition $\Gamma'''' \vdash \iota' : (\tau \oplus K'_{\triangleright})$. We conclude c) $\Gamma'''' \vdash e_3[x \leftarrow \iota'] : \delta$. So, from a), by Rule (Eval Reconfig), we have $e_3[x \leftarrow \iota']; S_{n+1}[\iota' \mapsto s'][\,\iota_i{\mapsto}v_i\ ^{i\in 1..n}] \downarrow v; S''''$, and with c), by induction hypothesis, we obtain the final result that there is a Γ'''' such that $\Gamma'''' \vdash v : \delta'$ with $\Gamma'''' \vdash \delta' \leq \delta$ and Γ'''' extends Γ''' and types S''''.

Finally, the occurrence of nil values in the heap follows the reasoning used in the instantiation process. In the first case, all nil values introduced by the new required ports are replaced by non-nil values, so we prove that $\mathsf{nil}(S'''') = \emptyset$.

***Case:* (Eval Reconfig Else)**

The second evaluation possibility for a reconfiguration follows without applying the configurator to the instance and therefore the typing of the resulting value v results by induction hypothesis on the typing derivation of e_4.

***Case:* (Eval Type Application)**

If the last rule is (Eval Type Application) the for the expression $e\langle\tau\rangle$ we have the typing judgement, whose premise must be a) $\Delta \vdash e : \forall_{X \leq \tau}\, \sigma$ and the evaluation judgement b) $e\langle\tau\rangle; S \downarrow v; S''$. The premises for b) are, c) $e; S \downarrow \Lambda X \leq \tau.e'; S'$ and d) $e'[X \leftarrow \tau']; S' \downarrow v; S''$. By induction hypothesis we know that there is a Γ' which extends Γ and types S' and $\Gamma' \vdash \Lambda X \leq \tau.e' : \forall_{X \leq \tau}\, \sigma'$ such that $\Gamma' \vdash \forall_{X \leq \tau}\, \sigma \leq \forall_{X \leq \tau}\, \sigma'$. From the polymorphic case of S (Definition 6.12) we know that $\Gamma', X \leq \tau \vdash \sigma' \leq \sigma$. By Rule (Val Type Abstraction) on a) we know that $\Gamma', X \leq \tau \vdash e' : \sigma$. Since $\Gamma \vdash \tau' \leq \tau$, by Lemma 6.20 (weakening) we have $\Gamma' \vdash \tau' \leq \tau$, and by Lemma 6.19 (substitution) we have that e) $\Gamma' \vdash e'[X \leftarrow \tau'] : \sigma'[X \leftarrow \tau']$ and by Lemma 6.18 (substitution) we have that $\Gamma' \vdash \sigma'[X \leftarrow \tau'] \leq \sigma[X \leftarrow \tau']$.

From e) and d), by induction hypothesis, we have that there is a Γ'' that extends Γ and types S'', such that $\Gamma'' \vdash v : \sigma''$ with $\Gamma'' \vdash \sigma'' \leq \sigma'[X \leftarrow \tau']$ and v is one of the possible values admitted in the lemma. The induction hypothesis on both cases indicates that $\mathsf{nil}(S'') = \emptyset$. Notice that v and S''' are record-based.

When composition operations are involved the reasoning is also similar, but handling different value forms and using Rule (Val Configurator Value) to correctly type the results.

***Case:* (Eval Requires)$^{\rho}$**

The typing hypothesis for this case is $\Gamma \vdash (\mathsf{requires}\ \ell : I) : \sigma$ with $\sigma = \emptyset \Longrightarrow \{\ell \bullet I, \ell \triangleleft I\}$ where the expression $\mathsf{requires}\ \ell : \tau$ evaluates to a configurator. By Rule (Val Configurator Value)$^{\rho}$, we have $\Gamma' \vdash \mathsf{conf}(\sigma, \mathsf{requires}\ \ell : I) : \sigma$ with $\Gamma' = \Gamma$ and $S' = S$ with $\mathsf{nil}(S') = \emptyset$.

***Case:* (Eval Provides)$^{\rho}$**

The typing hypothesis for this case is $\Gamma \vdash (\mathsf{provides}\ \ell : \tau) : \sigma$ with $\sigma = \emptyset \Longrightarrow \{\ell \circ \tau, \ell \triangleright \tau\}$ where the expression $\mathsf{requires}\ \ell : \tau$ evaluates to a configurator. By Rule (Val Configurator Value)$^{\rho}$, we have $\Gamma \vdash \mathsf{conf}(\sigma, \mathsf{provides}\ \ell : \tau) : \sigma$ with $\Gamma' = \Gamma$ and $S' = S$ with $\mathsf{nil}(S') = \emptyset$.

Case: **(Eval Plug)$^\rho$**

The typing hypothesis for this case is $\Gamma \vdash \mathsf{plug}\ (\pi_1 : \tau_1)\ \mathsf{into}\ (\pi_2 : \tau_2) : \sigma$ with $\sigma = (\{\pi_2 \circ \tau_2, \pi_1 \bullet \tau_1\} \Longrightarrow \{\pi_1 \bullet \tau_1\})$ where the expression $\mathsf{plug}\ (\pi_1 : \tau)\ \mathsf{into}\ (\pi_2 : \tau)$ evaluates to a configurator. By Rule (Val Configurator Value)$^\rho$, we have $\Gamma \vdash \mathsf{conf}(\sigma, \mathsf{plug}\ (\pi_1 : \tau_1)\ \mathsf{into}\ (\pi_2 : \tau_2)) : \sigma$ with $\Gamma' = \Gamma$ and $S' = S$ with $\mathsf{nil}(S') = \emptyset$.

Case: **(Eval Method Block)$^\rho$**

The typing hypothesis for this case is $\Gamma \vdash x_K[\ell_i : \tau_i = e_i\ ^{i \in 1..n}] : \sigma$ with $\sigma = K \Longrightarrow K, \{x \bullet \{\ell_i : \tau_i\ ^{i \in 1..n}\}\}$ where the expression $x_K[\ell_i : \tau_i = e_i\ ^{i \in 1..n}]$ evaluates to a configurator. By Rule (Val Configurator Value)$^\rho$, we have $\Gamma \vdash \mathsf{conf}(\sigma, x_K[\ell_i : \tau_i = e_i\ ^{i \in 1..n}]) : \sigma$ with $\Gamma' = \Gamma$ and $S' = S$ with $\mathsf{nil}(S') = \emptyset$.

Case: **(Eval Uses)$^\rho$**

The evaluation of this expression produces a configurator where e is replaced by its resulting value, $x[e : \tau \Rightarrow \sigma]; S \downarrow \mathsf{conf}(\delta, x[v : \tau \Rightarrow \sigma]); S'$. The typing judgment implies $\Gamma \vdash x[e : \tau \Rightarrow \sigma] : \emptyset \Longrightarrow K$ with $\tau = \{\ell^r_i : \tau_i\ ^{i \in 1..n}\}, \sigma = \{\ell^p_j : \sigma_j\ ^{j \in 1..m}\}$ and $K = \{x \bullet \sigma, x.\ell^r_i \circ \tau_i\ ^{i \in 1..n}, x.\ell^p_j \bullet \sigma_j\ ^{j \in 1..m}\}$.

We have that $|\Gamma| \vdash e : \tau \Rightarrow \sigma$ and $e; S \downarrow v; S'$, which by induction hypothesis on the first part of the lemma implies that there is a $|\Gamma'|$ extending $|\Gamma|$ such that $|\Gamma'| \vdash v : \tau' \Rightarrow \sigma'$ with $|\Gamma'| \vdash \tau' \Rightarrow \sigma' \leq \tau \Rightarrow \sigma$ and $\mathsf{nil}(S') = \emptyset$.

We conclude, by Rule (Val Configurator Value)$^\rho$, that $|\Gamma'| \vdash \mathsf{conf}(\delta, x[v : \tau]) : \emptyset \Longrightarrow K$ with $\delta = \emptyset \Longrightarrow K$ and by Lemma 4.10 (weakening) that there is a Γ' extending Γ and $\Gamma' \vdash \mathsf{conf}(\delta, x[v : \tau]) : \emptyset \Longrightarrow K$.

Case: **(Eval Sequence)**

By induction hypothesis we reach the conclusion that there is a Γ' extending Γ and typing S' such that $\Gamma' \vdash \mathsf{conf}(\sigma, c_1) : \sigma$ with $\sigma = (K \Longrightarrow K', K_c)$ and $\mathsf{nil}(S') = \emptyset$. Also by induction hypothesis, there is Γ'' extending both Γ' and Γ and typing S'' such that $\Gamma'' \vdash \mathsf{conf}(\sigma', c_2) : \sigma'$ with $\sigma' = (K_c, K'' \Longrightarrow K''')$ and $\mathsf{nil}(S'') = \emptyset$. By Rule (Val Configurator Value) we conclude that $\Gamma'' \vdash \mathsf{conf}(\sigma'', (c_1; c_2)) : \sigma''$ with $\sigma'' = K, K'' \Longrightarrow K', K'''$.

We follow by proving that well-typed expressions never evaluate to wrong.

Case: **(Wrong Call)**

For an application to be well-typed we must have $\Gamma \vdash e_1(e_2) : \sigma$. Then, by inspection of the type system, we conclude that the only typing rule that may derive this judgment is Rule (Val Application), and therefore we have a) $\Gamma \vdash e_1 : \tau \rightarrow \sigma$ and b) $\Gamma \vdash e_2 : \sigma$. We also have that c) $e_1; S \downarrow v; S'$ which by induction hypothesis lets us conclude that there is a Γ' typing

S' such that $\mathit{\Gamma} \vdash v : \tau \to \sigma$ and v is an abstraction. Therefore if an application expression is well-typed, the Rule (Wrong Call) is never applicable.

Case: **(Wrong Assign)**

If $\mathit{\Gamma} \vdash e_1.\ell := e_2 : \tau$ then by Rule (Eval Assign) we know that a) $\mathit{\Gamma} \vdash e_1 : \{\!| \ldots, \ell : \tau, \ldots |\!\}$. The evaluation hypothesis has the premise b) $e_1; S \downarrow v; S'$. Taking a) and b), by induction hypothesis we know that there is $\mathit{\Gamma}'$ typing S' and extending $\mathit{\Gamma}$ such that $\mathit{\Gamma} \vdash v : \{\!| \ldots, \ell : \tau, \ldots |\!\}$ and v must be a location leading to a record. Thus Rule (Wrong Assign) is never applicable to well-typed assignment expressions.

Case: **(Wrong Assign 2)**

Following the reasoning from the previous case, we have $\mathit{\Gamma} \vdash \imath : \{\!| \ldots, \ell : \tau, \ldots |\!\}$ and that $\imath$ is a location leading to a record. By Definition 2.21 we know that $\mathit{\Gamma} \vdash S(\imath) : \{\!| \ldots, \ell : \tau, \ldots |\!\}$ which by Rule (Val Record) must contain the label ℓ and therefore Rule (Wrong Assign 2) is also never applicable.

Case: **(Wrong Select)**. Similar.

Case: **(Wrong Select 2)**. Similar.

Case: **(Wrong Compose)**

If $\mathit{\Gamma} \vdash \mathsf{compose}\ e : K_{\triangleleft} \Rightarrow K_{\triangleright}$ then we know that $\mathit{\Gamma} \vdash e : \emptyset \Longrightarrow K$ by induction hypothesis we have a $\mathit{\Gamma}'$ extends $\mathit{\Gamma}$ and types S' and that v is a value such that $\mathit{\Gamma} \vdash v : \emptyset \Longrightarrow K$. This value can only be a configurator value. So wrong is never issued by Rule (Wrong Compose).

Case: **(Wrong New)**

If $\mathit{\Gamma} \vdash \mathsf{new}\ e\ \mathsf{with}\ \ell^r_j := e_j\ ^{j \in 1..m} : \sigma$, by Rule (Val New), we have that $\mathit{\Gamma} \vdash e : \tau \Rightarrow \sigma$ and that $e; S \downarrow v; S'$. By induction hypothesis we obtain that there is a $\mathit{\Gamma}'$ extending $\mathit{\Gamma}$ and typing S' such that v is a component value (other possible values have different types). This is contradicts the application of (Wrong New).

Case: **(Wrong Sequence)**

If $\mathit{\Gamma} \vdash e_1; e_2 : K, K'' \Longrightarrow K', K'''$ then $\mathit{\Gamma} \vdash e_1 : K \Longrightarrow K', K_c$ and, as $e_1; S \downarrow v; S$ we know, by induction hypothesis, that there is $\mathit{\Gamma}'$ extending $\mathit{\Gamma}$ and typing S' such that v is a configurator value with $\mathit{\Gamma}' \vdash v : K \Longrightarrow K', K_c$. Once again, we find a contradiction.

Case: **(Wrong Sequence 2)**. Similar.

Case: **(Wrong Reconfig)**

By induction hypothesis on the premises typing and evaluating e_1, we know that there is a $\mathit{\Gamma}'$ that extends $\mathit{\Gamma}$ and types S' such that e_1 evaluates to a value v typed by $\mathit{\Gamma}' \vdash v : K \Longrightarrow K'$ that is either an abstraction, a component, a configurator, or a location that *refers-to* a record. By our typing relation we conclude that this is a configurator $\mathsf{conf}(K \Longrightarrow K', c)$. So, Rule (Wrong Reconfig) is never applicable.

Case: **(Wrong Reconfig 2)** and **(Wrong Reconfig 3)**

From the previous case we know that the value to which e_1 evaluates is a configurator, and if e_2 is typed by an interface, and $e_2; S' \downarrow v; S_0$. by induction hypothesis we know that it must be a location that is either *undefined* or *refers-to* a record. So, Rules (Wrong Reconfig 2) and (Wrong Reconfig 3) are never applied.

Case: **(Wrong Type App)**

In this case we have that $e\langle\tau\rangle; S \downarrow \mathsf{wrong}; S'$ because $e; S \downarrow v; S'$ and $v \neq \Lambda X \leq \tau'.e'$. We also know that the expression is well typed, i.e. $\Delta \vdash e\langle\tau'\rangle : \sigma[X \leftarrow \tau']$ and by backward application of Rule (Val Type Application) we have $\Delta \vdash e : \forall_{X \leq \tau}\ \sigma$ with $\Delta \vdash \tau' \leq \tau$. By inspection of the type system we see that the only values that are typed with second-order types are type abstractions. Therefore, Rule (Wrong Type App) is never applicable on well-typed expressions.

The second part of the lemma states that the application of a configurator to an instance causes an effect consistent with the type of the operation. This proof is done by induction on the height of the derivations and by case analysis of the last application rule used. We verify that the type information in the instances is, at any time, sound with the global typing of the heap. We use the first part of the lemma when necessary.

Case: **(App Provides)**$^\rho$

In this case we have an expression typed $\Gamma \vdash (\mathsf{provides}\ \ell : \tau) : \emptyset \Longrightarrow \{\ell \circ \tau, \ell \triangleright \tau\}$ and an instance $s = (r, e, p)_\Pi$ that complies with $\emptyset$. The resulting instance, $s' = (r, e, p \oplus \{\ell = \imath\})_{\Pi, \imath:\tau}$, trivially complies, by Definition 3.23, with the resource set $\{\ell \circ \tau, \ell \triangleright \tau\}$ and the typing environment $\Gamma' = \Gamma, \imath : \tau$ which also types the heap $S' = S[\imath \mapsto \mathsf{nil}]$. The resulting value is well-typed according to Rule (Val Object).

Since the demanded resources in the configurator type is empty we have that $\mathsf{nil}(S) \subseteq X$ for some X. By Rule (App Provides)$^\rho$ we know that $\mathrm{select}_{S'}(s', \ell) = \imath$ and that $S'(\imath) = \mathsf{nil}$. More, $\imath$ is the only new location in S' with relation to S and $K'_\triangleleft \cup K'_\circ = \{\ell : \tau\}$. So, $\{\mathrm{select}_{S'}(s', \ell) \mid (\ell : \tau) \in K'_\triangleleft \cup K'_\circ\} = \{\imath\}$ and therefore $\mathsf{nil}(S') \subseteq \{\imath\} \cup X$ which corresponds to the expected results. With relation to the partially linked object type, we have that if the initial type of s is $[\![\tau \Rightarrow \sigma]\!]$ then the partial type of s' is $[\![\tau \Rightarrow \sigma \oplus \{\ell : \tau\}]\!]$.

Case: **(App Requires)**$^\rho$

In this case we have the typing judgment $\Gamma \vdash (\mathsf{requires}\ \ell : \tau) : \emptyset \Longrightarrow \{\ell \bullet \tau, \ell \triangleleft \tau\}$ as hypothesis and an instance $s = (r, e, p)_\Pi$ that complies with $\emptyset$. The resulting instance, $s' = (r \oplus \{\ell =$

$\imath\}, e, p)_{\Pi,\imath:\tau}$, trivially complies, by Definition 3.23, with the resource set $\{\ell \bullet \tau, \ell \lhd \tau\}$ and the typing environment $\Gamma' = \Gamma, \imath : \tau$, which also types the heap $S' = S[\imath \mapsto \mathsf{nil}]$. The resulting value is well-typed according to Rule (Val Object).

Since the demanded resources in the configurator type is empty we have that $\mathsf{nil}(S) \subseteq X$ for some X. By Rule (App Requires)$^\rho$ we know that $\mathrm{select}_{S'}(s', \ell) = \imath$ and that $S'(\imath) = \mathsf{nil}$. More, $\imath$ is the only new location in S' with relation to S and $K'_\lhd \cup K'_\circ = \{\ell : \tau\}$. So, $\{\mathrm{select}_{S'}(s', \ell) \mid (\ell : \tau) \in K'_\lhd \cup K'_\circ\} = \{\imath\}$ and therefore $\mathsf{nil}(S') \subseteq \{\imath\} \cup X$ which corresponds to the expected results. With relation to the partially linked object type, we have that if the initial type of s is $[\![\tau \Rightarrow \sigma]\!]$ then the partial type of s' is $[\![\tau \oplus \{\ell : \tau\} \Rightarrow \sigma]\!]$.

***Case:* (App Uses)$^\rho$**

If the last evaluation rule is (App Uses)$^\rho$ then we have a) $s; x[v : \tau]; S \Downarrow s'; S'$ and also
b) $\Gamma \vdash x[v : \tau] : \emptyset \Longrightarrow K$ with $K = \{x \bullet \{\ell^p_j : \sigma'_j\ {}^{j\in 1..m}\}, x.\ell^r_i \circ \tau'_i\ {}^{i\in 1..n}, x.\ell^p_j \bullet \sigma'_j\ {}^{j\in 1..m}\}$ where
$\tau = \{\ell^r_i : \tau_i\ {}^{i\in 1..k}\} \Rightarrow \{\ell^p_j : \sigma_j\ {}^{j\in 1..m}\}$.

From b), by Rule (Comp Uses), we have that $|\Gamma| \vdash v : \tau$, and since the only kind of values typed in this way are components, we also have that $v = \mathsf{comp}(c)$ with $|\Gamma| \vdash \mathsf{comp}(c) : \tau$. By Rule (Val Composition Value) we known that $|\Gamma| \vdash \mathsf{compose}\ c : \tau$ and from Rule (Val Compose) we know c) $|\Gamma| \vdash c : \emptyset \Longrightarrow K'$ with $K'_\circ = \emptyset$ and $\tau = (K'_\lhd \Rightarrow K'_\rhd)$.

From a), by Rule (App Uses)$^\rho$, we know that $(\mathsf{new}\ v); S \downarrow \imath; S'$. Since v evaluates to itself causing no changes to the heap, Rule (Eval Value), and there are no plug assignments to be considered, the evaluation of $\mathsf{new}\ v$ is supported, by Rule (Eval New), on d) $\mathbf{0}; c; S \Downarrow s''; S''$ with $S' = S''[\imath \mapsto s'']$.

Since the set of demanded resources in the type assigned in b) ($\emptyset \Longrightarrow K$) is empty, we know, from the conditions of the lemma, that there is a set X such that $\mathsf{nil}(S) \subseteq X$. From c) and d), by induction hypothesis on the second case of the lemma, we have that there is a $|\Gamma'|$ that types S'' and that the resulting value, s'', is compliant with K' with the partially linked object type $[\![K'_\lhd \Longrightarrow K'_\rhd]\!]$ with relation to $|\Gamma'|$. Remember that the initial instance is $\mathbf{0}$ and therefore its partial object type is $[\![\{\} \Rightarrow \{\}]\!]$.

The instance resulting from applying $x[v : \tau]$ to s is therefore $s' = (r, e \oplus \{x = \imath\}, p)_{\Pi,\imath:K'_\rhd}$, see (App Uses), and the resulting heap is S'. They are both well typed with relation to $|\Gamma''| = |\Gamma'|, \imath : K'_\rhd$ and by weakening with relation to Γ'' (by adding the elements taken from Γ to produce $|\Gamma|$).

More, the instance $(r, e \oplus \{x = \imath\}, p)_{\Pi,\imath:K'_\rhd}$ is, according to Definition 3.23, compliant with K. The induction hypothesis also tells us that $\mathsf{nil}(S'') \subseteq \{\mathrm{select}_{S''}(s, \ell) \mid (\ell : \tau) \in (\ell^r_i : \tau_i)^{i\in 1..n}\} \cup X$ with $K'_\lhd = \{\ell^r_i : \tau_i\ {}^{i\in 1..k}\}$. Which, in the involving composition context, can be written by dereferrencing x, $\mathsf{nil}(S') \subseteq \{\mathrm{select}_{S''}(s', \pi) \mid (\pi : \tau) \in K_\circ\} \cup X$ where $K_\circ = \{x.\ell^r_i : \tau_i\ {}^{i\in 1..n}\}$.

The partial type of the resulting object remains unchanged since no required or provided ports are added to the instance.

Case: **(App Method Block)**

If $\Gamma \vdash x_K[\ell_i : \tau_i = v_i\ ^{i\in 1..n}] : K \Longrightarrow K, \{x \bullet \{\ell_i : \tau_i\ ^{i\in 1..n}\}\}$, then by Rule (Comp Method Block), we know that for all $i \in 1..n$, $|\Gamma|, x : \{\!|\ell_i : \tau_i\ ^{i\in 1..n}|\!\}, K_\bullet \vdash v_i : \tau_i$. Since $(r, e, p)_\Pi$ is compliant with K and $v_i[(r, e, p)_\Pi][x \leftarrow \iota]$ denotes the substitution of the available names in the current instance $(r, e, p)_\Pi$ by its locations, which are typed in Γ, by Lemma 4.10 (weakening) and Lemma 4.11 (substitution), we obtain $\Gamma, x : \{\!|\ell_i : \tau_i\ ^{i\in 1..n}|\!\} \vdash v_i[(r, e, p)_\Pi] : \tau_i$ for all $i \in 1..n$ ($|\Gamma|$ correctly types the instance). Let $\Gamma' = \Gamma, \iota : \{\!|\ell_i : \tau_i\ ^{i\in 1..n}|\!\}$ and by Lemma 4.10 (weakening) and 4.11 (substitution) we have that $\Gamma' \vdash v_i[(r, e, p)][x \leftarrow \iota] : \tau_i$ for all $i \in 1..n$. So, we have that $\Gamma', \iota_i : \tau_i\ ^{i\in 1..n}$ types the resulting heap $S' = S[\iota \mapsto \{\ell_i = \iota_i\ ^{i\in 1..n}\}][\iota_i \mapsto v_i[(r, e, p)][x \leftarrow \iota]\ ^{i\in 1..n}]$.

Since $(r, e, p)_\Pi$ is compliant with K this makes the instance, $(r, e \oplus \{x = \iota\}, p)$, compliant with $K' = K, \{x \bullet \{\ell_i : \tau_i\ ^{i\in 1..n}\}\}$. No references are created in the heap that lead to nil, hence $\mathsf{nil}(S') = \mathsf{nil}(S)$, and by considering the set X such that $\mathsf{nil}(S) \subseteq \{\mathrm{select}_S(s, \ell) \mid (\ell : \tau) \in K_\triangleleft \cup K_\circ\} \cup X$ we have that $\mathsf{nil}(S') \subseteq \{\mathrm{select}_S(s, \ell) \mid (\ell : \tau) \in K'_\triangleleft \cup K'_\circ\} \cup X$ which is the same set. As in the previous case, the partial object type remains unchanged.

Case: **(App Plug)**

If $\Gamma \vdash \mathsf{plug}\ (\pi_1 : \tau)\ \mathsf{into}\ (\pi_2 : \tau) : (\{\pi_2 \circ \tau, \pi_1 \bullet \tau\} \Longrightarrow \{\pi_1 \bullet \tau\})$ only the unsatisfied inner requirements change, So $K'_\circ \subseteq K_\circ$, with $K = \{\pi_2 \circ \tau, \pi_1 \bullet \tau\}$ and $K' = \{\pi_1 \bullet \tau\}$. However, the resulting instance is the same and by Definition 3.23, s is compliant with K'. On the other hand, the heap is changed to make the connection between the source and the target of the plug operation. From the lemma's hypothesis we know that $\mathsf{nil}(S) \subseteq \{\mathrm{select}_S(s, \ell) \mid (\ell : \tau) \in K_\triangleleft \cup K_\circ\} \cup X$. $S(\mathrm{select}_S(s, \pi_2)) = \mathsf{nil}$, hence $\mathsf{nil}(S[\mathrm{select}_S(s, \pi_2) \mapsto \mathrm{select}_S(s, \pi_1)]) \subseteq \{\mathrm{select}_S(s, \ell) \mid (\ell : \tau) \in K'_\triangleleft \cup K'_\circ\} \cup X$.

Case: **(App Sequence)**

We first prove that the resulting instance is compliant with the type. The typing judgement $\Delta \vdash e_1; e_2 : K, K'' \Longrightarrow K', K'''$ is supported by the premises $\Delta \vdash e_1 : K \Longrightarrow K', K_c$ and $\Delta \vdash e_2 : K_c', K'' \Longrightarrow K'''$. By induction hypothesis on the second case of the lemma, together with the application of e_1 to an instance s compliant with K we reach an instance s' which is compliant with K', K_c. Since s' extends s it is also compliant with K'' and by Lemma 4.10 (weakening) and induction hypothesis on the second case of the lemma together with the evaluation of e_2 we conclude that s'' is compliant with K'''. Since it extends s and s' we conclude that it is also compliant with K'.

Now concerning the nil values in the heap. In the case of $s; (c_1; c_2); S \Downarrow s''; S''$ with $\Delta \vdash (e_1; e_2) : K, K'' \Longrightarrow K', K'''$ we have that there is a set X such that $\mathsf{nil}(S) \subseteq \{\mathrm{select}_S(s, \ell) \mid (\ell : \tau) \in (K, K'')_\triangleleft \cup (K, K'')_\circ\} \cup X$. By rewriting this definition we obtain $\mathsf{nil}(S) \subseteq \{\mathrm{select}_S(s, \ell) \mid (\ell : \tau) \in K_\triangleleft \cup K_\circ\} \cup X'$ with $X' = \{\mathrm{select}_S(s, \ell) \mid (\ell : \tau) \in K''_\triangleleft \cup K''_\circ\} \cup X$. Given the application of the first premise $s; c_1; S \Downarrow s'; S'$ typed $\Delta \vdash e_1 : K \Longrightarrow K', K_c$, by induction hypothesis, we know that

$\mathsf{nil}(S') \subseteq \{\text{select}_S(s,\ell) \mid (\ell:\tau) \in (K',K_c)_{\triangleleft} \cup (K',K_c)_{\circ}\} \cup X'$. Which is the same as $\mathsf{nil}(S') \subseteq \{\text{select}_S(s,\ell) \mid (\ell:\tau) \in K'_{\triangleleft} \cup K'_{\circ}\} \cup \{\text{select}_S(s,\ell) \mid (\ell:\tau) \in K_{c\triangleleft} \cup K_{c\circ}\} \cup \{\text{select}_S(s,\ell) \mid (\ell:\tau) \in K''_{\triangleleft} \cup K''_{\circ}\} \cup X$. Again, by rearranging the definition we have $\mathsf{nil}(S') \subseteq \{\text{select}_S(s,\ell) \mid (\ell:\tau) \in K'',K_{c\triangleleft} \cup K'',K_{c\circ}\} \cup X''$ with $X'' = \{\text{select}_S(s,\ell) \mid (\ell:\tau) \in K'_{\triangleleft} \cup K'_{\circ}\} \cup X$. Given the second premise evaluated by $s;c_1;S \Downarrow s';S'$ and typed by $\Delta \vdash e_2 : K_c, K'' \Longrightarrow K'''$, by induction hypothesis we know that $\mathsf{nil}(S'') \subseteq \{\text{select}_S(s,\ell) \mid (\ell:\tau) \in K',K'''_{\triangleleft} \cup K',K'''_{\circ}\} \cup X''$. As the locations in X'' corresponding to the labels in K' are already in this final set we conclude that $\mathsf{nil}(S'') \subseteq \{\text{select}_S(s,\ell) \mid (\ell:\tau) \in K',K'''_{\triangleleft} \cup K',K'''_{\circ}\} \cup X$. □

Appendix B

ComponentJ

In this section we present the programming language componentJ by means of simple examples and make a parallel with the $\lambda_{\chi}^{\leq}$ model language whenever it is convenient. The general structure of a standalone componentJ file is a sequence of static declarations, which include: the import of Java types and exceptions, the declaration of port interfaces, component interfaces, and components, and an expression that gets executed in the context of these declarations. The complete syntax of componentJ is depicted in Appendix B.5.

B.1 Hello World

The first program we introduce here is the classic program "Hello World!". This example illustrates the basic construction, composition and scripting mechanisms at a small scale. We start by declaring a port interface IHello with a method hello returning a string:

```
port interface IHello {
  string hello();
}
```

Notice that a port interface declaration roughly corresponds to a Java interface and also that type **string** is primitive in componentJ. We then define of a component Hello which provides a service specified by interface IHello in a port p:

```
component Hello {
  provides IHello p;
  methods m {
    string hello() {
      return "Hello␣World!";
    }
  }
  plug m into p;
}
```

The actual implementation of the provided service is defined in the inner method block m which is connected to the declared provided port by a plug operation. The component Hello can then be instantiated to an instance of type

```
object interface THello {
  provides IHello p;
}
```

and its method hello in port p invoked in the following componentJ fragment:

```
THello o = new Hello;
o.p.hello()
```

Although this expression yields the expected string value it does not meet the expectations of a classic "Hello World!" example. componentJ does not have global variable definitions and does not provide primitive i/o operations. Therefore, we must rely on external functionality to actually produce some output. In the next excerpt, port interface IPrint specifies the needed functionality and component PrintHello adapts Hello defined above to print the message:

```
port interface IPrint {
  void print(string);
}
port interface IPrintHello {
  void hello();
}
component interface CHello {
  provides IHello p;
}
component PrintHello {
  requires IPrint out;
  provides IPrintHello p;
  uses CHello h = Hello;
  methods m {
    void hello() {
      out.print(h.p.hello());
    }
  }
  plug m into p;
}
```

Notice that component PrintHello we've just created has a required port out which refers to a not yet available implementation of port interface IPrint. It is used inside the component and will be available whenever the component's implementation is used. PrintHello is a compound component whose architecture integrates an inner component Hello and a method block for scripting code, in this case taking the result of method hello in port h.p and printing it in the port p. Method block m is linked to the provided port p in order to export it.

Now, to use PrintHello, we need to link its required port to a service that indeed provides a method print which receives a string as argument. To actually print the message in a console we use a native java component providing this functionality; consider that a component Console exists and that it provides a port out typed IPrint. We can then instantiate PrintHello in the following way:

```
c = new Console;
...
o = new PrintHello [c.out in out];
o.p.hello();
```

The expression **new** C[o **in** p] instantiates C and connects the reference o to the required port p. Hence, the plug assignment of port c.out, of a new Console object, to the required port out of the new instance of PrintHello, o, "configures" the behaviour of this particular instance. So, if Console indeed provides access to a system console the intended message gets printed.

Other configurations are possible depending on the functionality provided at instantiation time. For instance, we could use a component Dialog, also providing a port out with type IPrint, that opens a dialogue box in a graphic display and shows up the message. Then, the result of

```
d = new Dialog;
...
o = new PrintHello [d.out in out];
o.p.hello();
```

is a dialogue box on some screen yielding the "Hello World!" message.

B.2 Dynamic Composition

We now illustrate dynamic composition in componentJ by elaborating on the previous example. Our goal is to define a way of uniformly extending any console component, one that implements IPrint in a port out, with HTML formatting tools. Consider that the following interface:

```
port interface IHTML {
  void openTag(string);
  void closeTag(string);
}
```

represents the HTML formatting tools available. In the context of an HTML editor, one can design a component that accepts plain character inputs in one port (to be called by a keyboard event handler) and formatting commands in another (to be connected to a toolbar event handler):

```
component interface CHTMLConsole{
  provides IHTML html;
```

```
    provides IPrint out;
  }
```

We can then define a method that takes a console component and produces another component which combines basic character input with formatting. This factory method is defined in the following port interface:

```
  port interface IHTMLWrapper {
    CHTMLConsole wrap(CConsole);
  }
```

and the corresponding implementation is as follows:

```
  component HTMLWrapper {
    provides IHTMLWrapper p;
    methods m {
      CHTMLConsole wrap(CConsole C) {
        return compose {
          provides IHTML html;
          provides IPrint out;
          uses CConsole c = C;
          methods m {
            void openTag(string t) {c.out.print("<"+t+">");}
            void closeTag(string t) {c.out.print("</"+t+">");}
          }
          plug c.out into out;
          plug m into html;
        };
      }
    }
    plug m into p;
  }
```

Notice that method wrap returns a component value whose inner element C is only instantiated at runtime. Any basic console component passed to this method is wrapped in a new composition implementing the formatting functionality by encoding HTML code in the output text.

Now, and given a component interface CHTMLConsole and an object interface HTMLConsole to type such components and instances:

```
  object interface CHTMLConsole {
    provides IHTML html;
    provides IPrint out;
  }
  object interface THTMLConsole {
    provides IHTML html;
    provides IPrint out;
  }
```

The following fragment would simply display our greeting message with the expected formatting.

```
CHTMLWrapper w = new HTMLWrapper;
CHTMLConsole c = w.p.wrap(Console);
THTMLConsole o = new c;
o.html.openTag(”html”);
o.html.openTag(”h1”);
o.out.print(”Hello␣World!”);
o.html.closeTag(”h1”);
o.html.closeTag(”html”);
```

In a text based console the result would be “`<html><h1>Hello World!</h1></html>`”.

These examples illustrate the usage of componentJ in the definition of both static and dynamic compositions. Although very simple they capture the essential aspects of the component programming in componentJ. Some more examples are also available in the componentJ distribution package.

We now describe the interoperability mechanisms between Java and componentJ.

B.3 Interoperability with Java

The integration of componentJ with the Java platform, and in particular with Java natively defined components, is achieved by means of a set of a minimal run-time support and by a set of classes and interfaces automatically generated by the componentJ compiler based on componentJ declarations.

Our prototype compiler works by generating the java code that manages component values at run-time; this involves declaring and creating new component instances and being able to combine components at run-time. We map componentJ entities as closer as possible to Java counterparts, e.g. port types are mapped directly to java interfaces, method blocks are mapped to classes and component instances are mapped to objects that give access to other objects (method blocks) by dereferencing a port name. The successive port connections are resolved at instantiation-time so that methods are at the distance of one dereferenciation step. However, not every componentJ entities have a Java counterpart as it is the case of components and component types: components are implemented in singleton classes with factory methods for the instantiation process, and component values are all typed alike in the context of Java programs. Hence, explicit type conversions and wrapping are necessary to use them safely.

From componentJ to Java Our prototype compiler uses componentJ declarations to generate interconnection code; it generates *client stubs* from component types to allow a Java program to apply type coercions to components and instances, thus accessing their ports in a type safe way. Component types are also used to produce *skeleton classes* to help writing native Java components that conform to a particular type. A skeleton is an abstract class implementing all the needed functionality, thus allowing for a fully operational native component to be implemented by extending such a skeleton and adding the intended Java code in the subclass. An empty example of a component class is also generated automatically to illustrate a few conventions.

From Java to ComponentJ Java types and exceptions are definitely useful when defining programs using the two languages. As componentJ explicitly excludes the notion of class (a type attached to one particular implementation) we chose to import java classes and interfaces (which may use class names in their definition) as opaque types explicitly imported by the programmer. As the current implementation of componentJ does not implement any notion of packages (for components), this type import facility also allows for the usage of types stored in packages. A local alias is created in order to use them in componentJ programs. The same happens with exceptions which, from the point of view of componentJ, can only be used in the method headers. There is no explicit exception handling in the current version of componentJ other than considering them in method typing.

Separate compilation is supported by the encoding of componentJ into Java by using the reflection mechanism of the JVM. Each type declaration and component declaration is therefore encoded in Java code and retrieved from its bytecode representation. Therefore, no particular associations between filenames and components is enforced. The name of a file is only used if it includes a componentJ final expression, in this case the name is used to create a class whose main function "is" the expression. In some cases, to retrieve type information not directly encodable in Java types we use explicit annotations on the generated Java entities.

B.4 Type System

componentJ type language includes a limited subset of the Java primitive types, interface types (called port interfaces), component types (component interfaces), and object types (object interfaces). It also includes arrays of any of such types above. Unbounded type abstraction is associated with port interfaces and bounded type abstraction is associated with components and component interfaces. The reason for this distinction is that components values can be polymorphic and ports can not. The instantiation of a component requires for its complete type instantiation and therefore ports are only manipulated after type instantiated.

For port interfaces, type equivalence (and subtyping) is based on the explicit hierarchy of types created by the explicit extension of port interfaces. At the level of objects and components we implement a structural subtyping relation miming the relation of $\lambda^{\leq}_{\chi}$: polymorphic component types are related according to the $F_{\leq}$ subtyping discipline (which is more general than the kernel-Fun discipline and is decidable because it is based on the name-based subtyping of port interfaces). Type recursion is also based on type names.

Consider an hypothetic port interface named Point, and the following samples of type declarations which illustrate type abstraction, type application and type extension mechanisms:

```
port interface IList<X> {
  void addFirst(X);
  X remove(int);
  X get(int);
}
port interface IListLast<X> extends IList<X> {
  void addLast(X);
}
component interface TPointList <X extends Point> {
  provides IList<X> list;
}
component interface TList <X> {
  provides IListLast<X> list;
  provides IIterator<X> iterator;
}
```

From these declarations the componentJ type system can derive that TList is a subtype of TPointList. Observe that, from the point of view of type safe substitution [], this relation makes sense: a list component of type TList, that can be parameterised with any type (not necessarily only objects representing points) and that provides more than the simple IList interface, fits in a context where a value of type TPointList is expected.

The implementation of a complete componentJ framework is an engineering problem beyond the scope of this work. It includes, besides the compiler: a run-time support infrastructure with global naming services, searching mechanisms based on typing, a supporting component library whose elements range from basic components to factory components that enforce and reuse programming patterns like the one illustrated in [83].

One aspect worth mentioning with relation to componentJ is the one that relates dynamic loading and strong typing. The dynamic loading of component values implies the crossing of the frontier between the untyped world of binary components and the strongly typed environment of componentJ programs. The identification, loading and authentication facilities provided by any underlying framework for componentJ are crucial for the success of such a language. Hence, any run-time infrastructure for componentJ should ensure at linking time that the actual type of the loaded component matches the type expected by the componentJ program.

Both of these aspects were out of the scope of this experiment.

B.5 Syntax

We here present the complete componentJ syntax. We use the usual extended BNF notation: for option (|), repetition (* and +), optional blocks (?). We present terminal nodes between quotes (””) and assume given non-terminals for identifiers (id), boolean, integer, float, and string literals. The repetition in simple name lists assumes comma separation (not in statement lists). The syntax is as follows:

```
Program ::= ((TypeImport|TypeDeclaration|ComponentDeclaration))* E?
TypeImport   ::= requires type Name as id
             |   requires exception Name as id
TypeDeclaration ::= port interface id (”<”IdList”>”)? ”=” TypeId ”;”
                |   port interface id (”<”IdList”>”)? extends TypeId* ”{”MethodHeader*”}”
                |   component interface id ”=” TypeId ”;”
                |   component interface id(”<”(id(extends Type)?)+”>”)? ”{”Port*”}”)
                |   object interface id ”{” ( ProvidedPort ”;” )*”}”
ComponentDeclaration  ::= component id ComposeExpressionBody
MethodHeader ::= Type id ”(” Type* ”)” ( throws id+ )? ”;”
Name  ::= id ( ”.” id )*
Type ::= ( PrimitiveType | TypeId ) (”[]”)?
PrimitiveType ::= void | int | short | long | boolean | float | double | string
TypeId  ::= id (”<”Type*”>”)?
S ::= E ”;” | Type id ( ”=” E )? ”;”
  | if (E) S (else S)? | while (E) S | ”{” S* ”}” | return (E)? ”;” | ”;”
E ::= E ”=” E | E ”||” E | E ”&&” E | ”!” E | E ”==” E | E ”!=” E | E ”<” E | E ”>” E
  | E ”<=” E | E ”>=” E | E ”+” E | E ”−” E | E ”*” E | E ”/” E | ”−” E | E ”%” E
  | id | E.id | E.id(E*) | E”[”E”]”
  | IntegerLiteral | FloatLiteral | BooleanLiteral | StringLiteral
  | new E”<”Type*”>” ”[” (E in id)* ”]”
  | compose ComposeExpressionBody
ComposeExpressionBody ::= (”<”(id(extends Type)?)+”>”)? ”{” CompositionOperation* ”}”
CompositionOperation  ::= PortDeclaration | Uses | MethodBlock | Plug
PortDeclaration ::= RequiredPort | ProvidedPort
RequiredPort ::= requires Type id ”;”
ProvidedPort ::= provides Type id ”;”
Uses  ::= uses Type id ”=” E ”;”
MethodBlock  ::= methods id ”{” ( Type id ”;” )* ( Method )*  ”}”
Plug ::= plug PlugName into PlugName ”;”
PlugName  ::= id ( ”.” id )?
Method ::= Type id ”(”id*”)” ( ”throws” Id+ )? ”{” S* ”}”
```

Bibliography

[1] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[2] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, April 1991.

[3] Martín Abadi and Marcelo P. Fiore. Syntactic considerations on recursive types. In *Proceedings of the Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 242–252. IEEE Computer Society Press, 1996.

[4] Jonathan Aldrich, Craig Chambers, and David Notkin. Architectural reasoning in archjava. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 334–367. Springer-Verlag, 2002.

[5] Joao Paulo A. Almeida, Maarten Wegdam, Marten van Sinderen, and Lambert Nieuwenhuis. Transparent dynamic reconfiguration for corba. In *Proceedings of the International Symposium on Distributed Objects and Applications*, pages 197–207, 2001.

[6] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993.

[7] Davide Ancona, Giovanni Lagorio, and Elena Zucca. JAM - A smooth extension of Java with mixins. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 154–178. Springer-Verlag, 2000.

[8] Davide Ancona, Giovanni Lagorio, and Elena Zucca. Smart modules for Java-like languages. In *ECOOP 2005 Workshop on Formal Techniques for Java-like Programs*, 2005.

[9] Davide Ancona and Eugenio Moggi. Program generation and components. In *Proceedings of the Formal Methods for Components and Objects (FMCO)*, number 3167. Springer-Verlag, 2004.

[10] Davide Ancona and Elena Zucca. A theory of mixin modules: Basic and derived operators. *Mathematical Structures in Computer Science*, 8(4):401–446, 1998.

[11] Davide Ancona and Elena Zucca. A calculus of module systems. *Journal of Functional Programming*, 12(2):91–132, 2002.

[12] Lorenzo Bettini, Viviana Bono, and Betti Venneri. Subtyping mobile classes and mixins. In *Proceedings of the International Workshop on Foundations of Object-Oriented Languages (FOOL)*, 2003.

[13] C. Bidan, V. Issarny, T. Saridakis, and A. Zarras. A dynamic reconfiguration service for corba. In *Proceedings of the International Conference on Configurable Distributed Systems*, pages 35–42, 1998.

[14] Gavin Bierman, Michael Hicks, Peter Sewell, and Gareth Stoyle. Formalizing dynamic software updating. In *Proceedings of the International Workshop on Unanticipated Software Evolution*, 2003.

[15] Viviana Bono, Amit Patel, and Vitaly Shmatikov. A core calculus of classes and mixins. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 43–66. Springer-Verlag, 1999.

[16] Gilad Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. PhD thesis, University of Utah, 1992.

[17] Gilad Bracha and Gary Lindstrom. Modularity meets inheritance. In *Proceedings of the IEEE Computer Society International Conference on Computer Languages*, pages 282–290, 1992.

[18] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 183–200, 1998.

[19] Michael Brandt and Fritz Henglein. Coinductive axiomatization of recursive type equality and subtyping. In *Proceedings of the International Conference on Typed Lambda Calculi and Applications (TLCA)*, volume 1210, pages 63–81. Springer-Verlag, 1997.

[20] Kim B. Bruce, Luca Cardelli, and Benjamin C. Pierce. Comparing object encodings. *Information and Computation*, 155(1/2):108–133, November 1999. Special issue of papers from *Theoretical Aspects of Computer Software (TACS 1997)*. An earlier version appeared as an invited lecture in the Third International Workshop on Foundations of Object Oriented Languages (FOOL 3), July 1996.

[21] Martin Büchi and Wolfgang Weck. Compound types for Java. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*, pages 362–373, 1998.

[22] L. Cardelli. Phase distinctions in type theory. Manuscript, 1988.

[23] Luca Cardelli. Amber. In Guy Cousineau, Pierre-Louis Curien, and Bernard Robinet, editors, *Combinators and functional programming languages : Thirteenth Spring School of the LITP, Val d'Ajol, France, May 6-10, 1985*, volume 242. Springer-Verlag, 1986. Also AT&T Bell Laboratories Technical Memorandum TM 11271-840924-10.

[24] Luca Cardelli. Program fragments, linking, and modularization. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 1997.

[25] Luca Cardelli. Type systems. In Allen B. Tucker, editor, *The Computer Science and Engineering Handbook*. CRC Press, Boca Raton, FL, 1997.

[26] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 report (revised). Technical report, Digital Systems Research Center, 1989.

[27] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.

[28] Giuseppe Castagna and Benjamin Pierce. Decidable bounded quantification. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, January 1994.

[29] Giuseppe Castagna and Benjamin Pierce. Corrigendum: Decidable bounded quantification. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 1995.

[30] Dario Colazzo and Giorgio Ghelli. Subtyping recursive types in Kernel Fun. In *Proceedings of the Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 137–146, 1999.

[31] Dario Colazzo and Giorgio Ghelli. Subtyping, recursion and parametric polymorphism in kernel fun. *Information and Computation*, 198(2):71–179, 2005.

[32] Microsoft Corporation. The COM+ platform. `http://www.microsoft.com/com/`.

[33] Karl Crary, Robert Harper, and Sidd Puri. What is a recursive module? In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 50–63, 1999.

[34] Pierre-Louis Curien and Giorgio Ghelli. Coherence of subsumption, minimum typing and type-checking in $F_{\leq}$. In *Theoretical aspects of object-oriented programming: types, semantics, and language design*, pages 247–292. MIT Press, 1994.

[35] Sophia Drossopoulou, Ferruccio Damiani, Mariangolia Dezani-Ciancaglini, and Paola Giannini. Fickle: Dynamic Object Re-Classification. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Springer-Verlag, 2001.

[36] Sophia Drossopoulou, Ferruccio Damiani, Mariangolia Dezani-Ciancaglini, and Paola Giannini. More dynamic object re-classification: Fickle II. *ACM Transactions on Programming Languages and Systems*, 2(24):153–191, 2002.

[37] Dominic Duggan. Type-based hot swapping of running modules. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 62–73, 2001.

[38] Dominic Duggan and Constantinos Sourelis. Mixin modules. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP)*, volume 31(6), pages 262–273, 1996.

[39] Miguel Durão. ComponentGlue. Master's thesis, Universidade Nova de Lisboa, 2005.

[40] M. Endler. A language for implementing generic dynamic reconfigurations of distributed programs. In *Proceedings of the Brazilian Symposium on Computer Networks*, 1994.

[41] Sonia Fagorzi and Elena Zucca. A calculus for reconfiguration. In *On-line prooceedings of International Workshop on Developments in Computational Models (DCM at ICALP05)*, 2005.

[42] Sonia Fagorzi and Elena Zucca. A calculus for reconfiguration. *under reviewing process for publication in Mathematical Structures in Computer Science*, 2005. eletronically available from the first author's web page.

[43] Robert Bruce Findler and Matthew Flatt. Modular object-oriented programming with units and mixins. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 94–104, 1999.

[44] Matthew Flatt and Matthias Felleisen. Units: Cool modules for HOT languages. *ACM SIGPLAN Notices*, 33(5):236–248, May 1998.

[45] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 171–183, 1998.

[46] I. R. Forman, M. H. Conner, S. H. Danforth, and L. K. Raper. Release-to-release binary compatibility in SOM. *ACM SIGPLAN Notices OOPSLA'95*, 30(10):426–438, 1995.

[47] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[48] Vladimir Gapeyev, Michael Levin, and Benjamin Pierce. Recursive subtyping revealed. *Journal of Functional Programming*, 12(6):511–548, 2003. Preliminary version appears in ICFP, 2000, with the title "Recusive types for Dummies". Also appears as Chapter 21 of [78].

[49] David Garlan. Software architecture: a roadmap. In *Proceedings of the ACM SIGSOFT International Conference on Software Engineering (ICSE)*, pages 91–101, 2000.

[50] Nadji Gauthier and François Pottier. Numbering matters: First-order canonical forms for second-order recursive types. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 150–161, 2004.

[51] Giorgio Ghelli. Recursive types are not conservative over $F_{\leq}$. In *Typed Lambda Calculus and Applications*. Springer-Verlag, 1993.

[52] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and types*. Cambridge University Press, 1989.

[53] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 123–137, 1994.

[54] Robert Harper, John C. Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 341–354, 1990.

[55] Michael W. Hicks, Jonathan T. Moore, and Scott Nettles. Dynamic software updating. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 13–23, 2001.

[56] Tom Hirschowitz and Xavier Leroy. Mixin modules in a call-by-value setting. In *Proceedings of the European Symposium on Programming (ESOP)*. Springer-Verlag, 2002.

[57] Tom Hirschowitz, Xavier Leroy, and J. B. Wells. Call-by-value mixin modules: Reduction semantics, side effects, types. In *Proceedings of the European Symposium on Programming (ESOP)*. Springer-Verlag, 2004.

[58] Christine R. Hofmeister. Dynamic reconfiguration of distributed applications. Technical Report CS-TR-3210, University of Maryland, 1994.

[59] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *ACM SIGPLAN Conference on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, October 1999. Full version in ACM Transactions on Programming Languages and Systems (TOPLAS), 23(3), May 2001.

[60] Alan Jeffrey. A symbolic labelled transition system for coinductive subytping of f-mu-sub types. In *Proceedings of the Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 323–333, 2001.

[61] Andrew Kennedy and Don Syme. Design and implementation of generics for the .net common language runtime. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 1–12, 2001.

[62] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall Professional Technical Reference, 1988.

[63] Fabio Kon, Binny Gill, Manish Anand, Roy H. Campbell, and M. Dennis Mickunas. Secure dynamic reconfiguration of scalable corba systems with mobile agents. In *Proceedings of the Second International Symposium on Agent Systems and Applications and Fourth International Symposium on Mobile Agents*, pages 86–98. Springer-Verlag, 2000.

[64] Fabio Kon, Manuel Román, Ping Liu, Jina Mao, Tomonori Yamane, Luiz Claudio Magalhães, and Roy H. Campbell. Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000)*, number 1795 in LNCS, pages 121–143. Springer-Verlag, 2000.

[65] Jeff Kramer. Configuration programming - a framework for the development of distributable systems. In *in Proceedings of IEEE International Conference on Computer Systems and Software Engineering*, 1990.

[66] Jeff Kramer and Jeff Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, 1990.

[67] Xavier Leroy. Manifest types, modules, and separate compilation. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 109–122, 1994.

[68] Xavier Leroy. A modular module system. *Journal of Functional Programming*, 10(3):269–303, 2000.

[69] Barbara Liskov. Keynote address - data abstraction and hierarchy. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*, pages 17–34, 1987.

[70] David B. MacQueen. Modules for standard ML. In *Proceedings of the 1984 ACM Conference on LISP and Functional Programming*, pages 198–207. ACM Press, 1984.

[71] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architectures. In *Proceedings of the European Software Engineering Conference (ESEC)*, pages 137–153. Springer-Verlag, 1995.

[72] Jeff Magee, Naranker Dulay, and Jeff Kramer. Regis: A constructive development environment for distributed systems. *in Distributed Systems Engineering Journal*, 5(1), 1994.

[73] J. Legatheaux Martins, Luís Caires, Sérgio Duarte Nuno Preguiça, João Costa Seco, and Henrique João Domingos. Databricks: Data components for mobile groupware. Electronically available at `http://asc.di.fct.unl.pt/databricks`.

[74] S. McDirmid, M. Flatt, and W. Hsieh. Jiazzi: New-age components for old-fashioned Java. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*, 2001.

[75] George C. Necula. Proof-carrying code. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 106–119, 1997.

[76] Martin Odersy. Programming in scala, 2006. Electronically available at `http://scala.epfl.ch/`.

[77] Benjamin C. Pierce. Bounded quantification is undecidable. In C. A. Gunter and J. C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, pages 427–459. MIT Press, 1994.

[78] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[79] Alastair Reid, Matthew Flatt, Leigh Stoller, Jay Lepreau, and Eric Eide. Knit: Component composition for systems software. In *Proc. of the 4th Operating Systems Design and Implementation (OSDI)*, pages 347–360, 2000.

[80] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew Black. Traits: Composable units of behavior. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Springer-Verlag, 2003.

[81] João Costa Seco and Luís Caires. A basic model of typed components. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Springer-Verlag, 2000.

[82] João Costa Seco. ML implementation of the subtyping algorithm on second-order equi-recursive types. Electronically available at `http://www-ctp.di.fct.unl.pt/~jcs/impl`.

[83] João Costa Seco. Adding type safety to component-oriented programming. In *OnLine Proceedings of the first PhD Student Workshop of FMOODS*, 2002.

[84] João Costa Seco. Type safe composition in .NET. In *First Microsoft Research Summer Workshop*, Cambridge, 2002.

[85] João Costa Seco and Luís Caires. Parametrically typed components. In *Workshop on Component Oriented Programming (WCOP), co-located with ECOOP'2000*, 2000.

[86] João Costa Seco and Luís Caires. ComponentJ: The reference manual. Technical Report UNL-DI-6-2002, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, 2002. A prototype compiler is electronically available at `http://ctp.di.fct.unl.pt/~jcs/ComponentJ/`.

[87] João Costa Seco and Luís Caires. Subtyping first-class polymorphic components. In *Proceedings of the European Symposium on Programming (ESOP)*, 2005.

[88] João Costa Seco and Luís Caires. Types for dynamic reconfiguration. In *Proceedings of the European Symposium on Programming (ESOP)*, 2006.

[89] Manuel Serrano. Wide classes. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Springer-Verlag, 1999.

[90] Peter Sewell. Modules, abstract types, and distributed versioning. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 236–247, 2001.

[91] Charles Smith and Sophia Drossopoulou. Chai : Traits for java-like languages. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Springer-Verlag, 2005.

[92] Vugranam C. Sreedhar. Mixin'up components. In *Proceedings of the ACM SIGSOFT International Conference on Software Engineering (ICSE)*, pages 198–207, 2002.

[93] Gareth Stoyle, Michael Hicks, Gavin Bierman, Peter Sewell, and Iulian Neamtiu. *Mutatis Mutandis*: Safe and Flexible Dynamic Software Updating. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 183–194, 2005.

[94] Christopher Strachey. Fundamental concepts in programming languages. *In Lecture notes of the International Summer School in Computer Programming*, Copenhagen, August 1967.

[95] Christopher Strachey. Fundamental concepts in programming languages. *High-Order and Symbolic Computation*, 13:11–49, 2000. A Reprint of [94].

[96] Sun Microsystems Inc. Enterprise JavaBeans specification, version 3.0. Electronically available at `http://java.sun.com/products/ejb/`.

[97] Clemens Szyperski. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley / ACM Press, 1998.

[98] Walid Taha and Tim Sheard. MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1-2):211–242, 2000.

[99] The Caml team. The OCaml language documentation. Electronically available at `http://www.ocaml.org`.

[100] A. van Wijngaarden, B.J. Mailloux, J.E.L. Peck, C.H.A. Koster, M. Sintzoff, C.H. Lindsey, L.G.T. Meertens, and R.G.Fisker. Revised Report on the Algorithmic Language ALGOL 68. Electronically available at `http://members.dokom.net/w.kloke/RR/`.

[101] Joe Wells and René Vestergaard. Confluent equational reasoning for linking with first-class primitive modules. In *Proceedings of the European Symposium on Programming (ESOP)*. Springer-Verlag, 1999.

[102] Michel Wermellinger and José Luiz Fiadeiro. Graph transformation approach to software architecture reconfiguration. *Science of Computer Programming*, 44(2), August 2002.

[103] Matthias Zenger. Type-safe prototype-based component evolution. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Springer-Verlag, 2002.